SECURING THE SOFTWARE SUPPLY CHAIN

RECOMMENDED PRACTICES GUIDE FOR

DEVELOPERS

Enduring Security Framework

August 2022

Executive Summary

Cyberattacks are conducted via cyberspace and target an enterprise's use of cyberspace for the purpose of disrupting, disabling, destroying, or maliciously controlling a computing environment or infrastructure; or destroying the integrity of the data or stealing controlled information.[1]

Recent cyberattacks such as those executed against SolarWinds and its customers, and exploits that take advantage of vulnerabilities such as Log4j, highlight weaknesses within software supply chains, an issue which spans both commercial and open source software and impacts both private and Government enterprises. Accordingly, there is an increased need for software supply chain security awareness and cognizance regarding the potential for software supply chains to be weaponized by nation state adversaries using similar tactics, techniques, and procedures (TTPs).

In response, the White House released an Executive Order on Improving the Nation's Cybersecurity (EO 14028). EO 14028 establishes new requirements to secure the federal government's software supply chain. These requirements involve systematic reviews, process improvements, and security standards for both software suppliers and developers, in addition to customers who acquire software for the Federal Government.

Similarly, the Enduring Security Framework[2] (ESF) Software Supply Chain Working Panel has established this guidance to serve as a compendium of suggested practices for developers, suppliers, and customer stakeholders to help ensure a more secure software supply chain. This guidance is organized into a three part series: Part 1 of the series focuses on software developers; Part 2 focuses on software suppliers; and Part 3 focuses on software customers.

Customers (acquiring organizations) may use this guidance as a basis of describing, assessing, and measuring security practices relative to the software lifecycle. Additionally, suggested practices listed herein may be applied across the acquisition, deployment, and operational phases of a software supply chain.

The software supplier (vendor) is responsible for liaising between the customer and software developer. Accordingly, vendor responsibilities include ensuring the integrity and security of software via contractual agreements, software releases and updates, notifications, and mitigations of vulnerabilities. This guidance contains recommended best practices and standards to aid suppliers in these tasks.

This document will provide guidance in line with industry best practices and principles which software developers are strongly encouraged to reference. These principles include security requirements planning, designing software architecture from a security perspective, adding security features, and maintaining the security of software and the underlying infrastructure (e.g., environments, source code review, testing).

---

[1] Committee on National Security Systems (CNSS)

[2] The ESF is a cross-sector working group that operates under the auspices of Critical Infrastructure Partnership Advisory Council (CIPAC) to address threats and risks to the security and stability of U.S. national security systems. It is comprised of experts from the U.S. government as well as representatives from the Information Technology, Communications, and the Defense Industrial Base sectors. The ESF is charged with bringing together representatives from private and public sectors to work on intelligence-driven, shared cybersecurity challenges.

DISCLAIMER

DISCLAIMER OF ENDORSEMENT

This document was written for general informational purposes only. It is intended to apply to a variety of factual circumstances and industry stakeholder, and the information provided herein is advisory in nature. The guidance in this document is provided "as is." Once published, the information within may not constitute the most up-to-date guidance or technical information. Accordingly, the document does not, and is not intended to, constitute compliance or legal advice. Readers should confer with their respective advisors and subject matter experts to obtain advice based on their individual circumstances. In no event shall the United States Government be liable for any damages arising in any way out of the use of or reliance on this guidance.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the United States Government, and this guidance shall not be used for advertising or product endorsement purposes. All trademarks are the property of their respective owners.

PURPOSE

NSA, ODNI, and CISA developed this document in furtherance of their respective cybersecurity missions, including their responsibilities to develop and issue cybersecurity recommendations and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

CONTACT

Client Requirements / Inquiries: Enduring Security Framework nsaesf@cyber.nsa.gov

Media Inquiries / Press Desk:

- NSA Media Relations, 443-634-0721, MediaRelations@nsa.gov
- CISA Media Relations, 703-235-2010, CISAMedia@cisa.dhs.gov
- ODNI Media Relations, dni-media@dni.gov

Table of Contents

1 Introduction

Unmitigated vulnerabilities in the software supply chain pose a significant risk to organizations. This paper presents actionable recommendations for a software supply chain's development, production and distribution, and management processes, to increase the resiliency of these processes against compromise.

All organizations have a responsibility to establish software supply chain security practices to mitigate risks, but the organization's role in the software supply chain lifecycle determines the shape and scope of this responsibility.

Because the considerations for securing the software supply chain vary based on the role an organization plays in the supply chain, this series presents recommendations geared toward these important roles, namely, developers, suppliers, and customers (or the organization acquiring a software product).

This guidance is organized into a three-part series and will be released coinciding with the software supply chain lifecycle. This is Part 1 of the series which focuses on software developers. Part 2 of the series focuses on the software supplier and Part 3 of the series focuses on the software customer. This series will help foster communication between these three different roles and among cybersecurity professionals that may facilitate increased resiliency and security in the software supply chain process.

In this series, terms such as risk, threat, exploit, and vulnerability are based on descriptions defined in Committee on National Security Systems Glossary (CNSSI 4009).[3]

1.1 Background

Historically, software supply chain compromises largely targeted commonly known vulnerabilities organizations that were left unpatched. While threat actors still use this tactic to compromise unpatched systems, a new, less conspicuous method of compromise also threatens software supply chains and undermines trust in the patching systems themselves that are critical to guarding against legacy compromises. Rather than waiting for public vulnerability disclosures, threat actors proactively inject malicious code into products that are then legitimately distributed downstream through the global supply chain. Over the last few years, these next-generation software supply chain compromises have significantly increased for both open source and commercial software products.

Technology consumers generally manage software downloads and broader, more traditional software supply chain activities separately. Considering both the upstream and downstream phases of software as a component of supply chain risk management may help to identify problems and provide a better way forward in terms of integrating activities to achieve systemic security. However, there are also some differences to account for in the case of software products. A traditional software supply chain cycle is from point of origin to point of consumption and generally enables a customer to return a malfunctioning product and confine any impact. In contrast, if a

[3] CNSSI-4009.pdf

software package is injected with malicious code which proliferates to multiple consumers; the scale may be more difficult to confine and may cause an exponentially greater impact.

Common methods of compromise used against software supply chains include exploitation of software design flaws, incorporation of vulnerable third-party components into a software product, infiltration of the supplier's network with malicious code prior to the final software product being delivered, and injection of malicious software that is then deployed by the customer.

Stakeholders must seek to mitigate security concerns specific to their area of responsibility. However, other concerns may require a mitigation approach that dictates a dependency on another stakeholder or a shared responsibility by multiple stakeholders. Dependencies that are inadequately communicated or addressed may lead to vulnerabilities and the potential for compromise.

Areas where these types of vulnerabilities may exist include:

- Undocumented features or risky functionality,
- Unknown and/or revisions to contractual, functionality or security assumptions between evaluation and deployment,
- Supplier's change of ownership and/or of geo-location, and
- Poor supplier enterprise or development hygiene.

1.2 Document overview

This document contains the following additional sections and appendices:

Section 2 recommends principles Developers may use to help secure the software development lifecycle (SDLC), an important process used to protect the software supply pipeline.

Section 3 is a collection of appendices supplementing the preceding sections:

Appendix A: Crosswalk Between the *NIST SP800-218; Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework* (SSDF)[4] and Use Cases described herein.

Appendix B: Dependencies

Appendix C: Supply-Chain Levels for Software Artifacts (SLSA)[5]

Appendix D: Recommended Artifacts and Checklist

Appendix E: Informative References

Appendix F: Acronyms

Each section contains examples of threat scenarios and recommended mitigations**.** Threat scenarios explain how processes that compose a given phase of the software development lifecycle (SDLC) relate to common vulnerabilities that could be exploited. The recommended mitigations present controls and mitigations that could reduce the impact of the threats.

[4] Draft NIST SP 800-218, Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities

[5] GitHub - slsa-framework/slsa: Supply-chain Levels for Software Artifacts

2 Developer

The secure software development lifecycle (Secure SDLC) is an important process used to secure the software supply chain. An example of the individual group activities and the relationships between the customers, developers and suppliers are represented in the figure below:

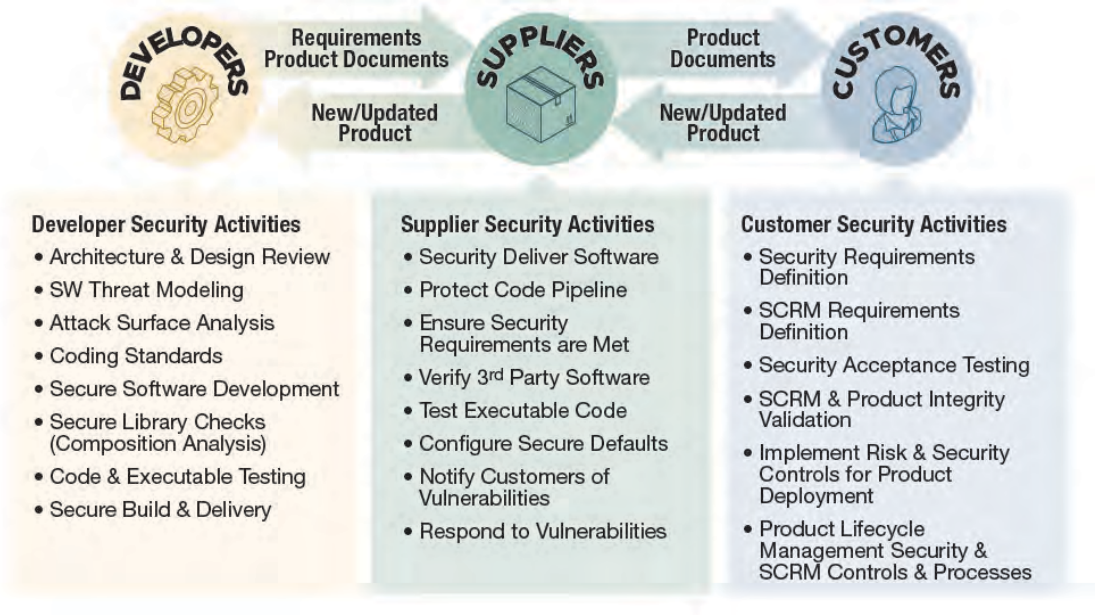

Figure 1: Software Supply Chain Group Relationships and Activities

The process starts when the supplier's program management team collects feature requests from their customer's user-base, technical base, and marketing teams. These features include both operational and security enhancements to the product and are used to generate use cases that are then formulated into prioritized requirements. The supplier and developer management teams work together to define the requirements that are used to produce the architecture and high-level design which a development team uses to produce a product. In addition, the combined management team defines the product development security policies and practices that are used when producing the product. The process defines how development activities will be structured and what artifacts will be collected for verification and validation. The following is a short list of examples of the Secure SDLC process and practices:

- NIST "Secure Software Development Framework,"[6]
- Carnegie Mellon University "Secure Software Development Lifecycle Processes,"[7]
- ACM "The Protection of Information in Computer Systems,"[8]
- OWASP "Secure Development Lifecycle,"[9]

[6] NIST Secure Software Development Framework
[7] Carnegie Mellon University Secure Software Development Lifecycle Processes
[8] https://web.mit.edu/Saltzer/www/publications/protection/
[9] OWASP Secure Software Development Lifecycle (SSDLC)

- "Cisco Secure Development Lifecycle,"[10]
- Synopsys "Secure Software Development Lifecycle Phases,"[11]
- US-Cert "Secure Software Development Lifecycle Processes,"[12]
- OpenSSF "Secure Software Development Fundamentals Courses."[13]

In addition to the high-level development documents produced, the management team defines the security practices and procedures used for secure software development such as:

- Secure coding practices,
- The code review process,
- Software repository procedures, testing, and vulnerability assessments,
- Procedures for securely building and distributing the product.

Once released, a product is monitored for defects through a support channel, available to product customers, and developers can securely provide updates and upgrades to address reported issues. For each operation within the Secure SDLC, artifacts are created which attest to the adherence to the processes required and outlined. These artifacts are outline in "**Appendix D: Artifacts and Checklist**."

2.1 Secure product criteria and management

As described in **Section 2. 2 Develop secure code** through **Section 2.5 Deliver code**, the developer use cases are dependent on the procedures and policies defined within a Secure SDLC process. Development team managers and members adapt and customize this process to meet their specific needs. The Secure SDLC identifies the exact procedures and policies that are used to ensure that secure development practices are implemented and artifacts are created to attest to the adherence of the adopted Secure SDLC plan with respect to the implementation and distribution of the product.

A development team is comprised of experts in development, quality assurance (QA), build engineering, and security. The product management team is comprised of individuals with product leadership experience, and includes product and development managers, security architects, and company-level quality control assessors, all contributing to product release oversight.

The top-level organizational management team must ensure secure development policies and procedures are supported within the budget and schedule and are implemented and adhered to by the assigned development teams. The figure below outlines a secure development process and lifecycle.

[10] Cisco Secure Development Lifecycle
[11] Synopsys Secure Software Development Lifecycle Phases
[12] US-Cert Secure Software Development Lifecycle Processes
[13] Secure Software Development Fundamentals Courses

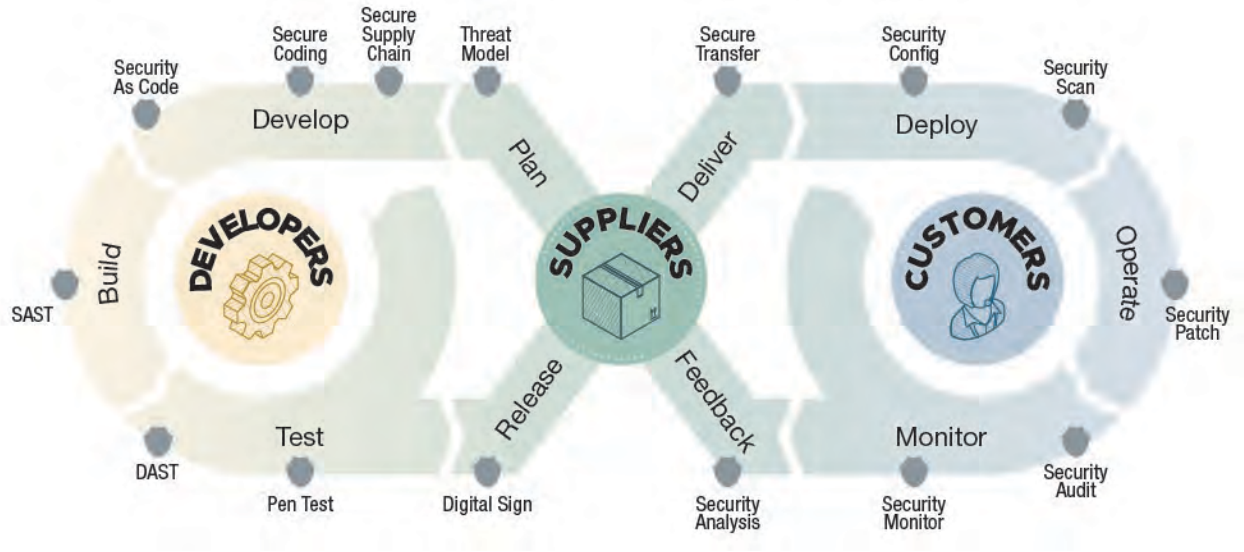

Figure 2: Secure Software Development Process (DoD Chief Information Officer, 2021)[14]

The example process illustrated above ensures that secure, resilient products are developed. It also illustrates that the development process can be measured using well-defined, tangible artifacts that may be collected, evaluated, and recorded to validate the use of the documented secure principles and guidelines outlined by the product management team.

Threat scenarios

When developing and delivering a product, the following common threats may occur during the software development lifecycle:

1. Adversary intentionally injecting malicious code or a developer unintentionally including vulnerable code within a product.
2. Incorporating vulnerable third-party source code or binaries within a product either knowingly or unknowingly.
3. Exploiting weaknesses within the build process used to inject malicious software within a component of a product.
4. Modifying a product within the delivery mechanism, resulting in injection of malicious software within the original package, update, or upgrade bundle deployed by the customer.

For more information on each threat scenario refer to Section **2.2 Develop Secure Code** through **Section 2.5 Deliver Code.** These sections contain more details for the threat scenarios and define strategies for each of the types of incidents or compromises that can occur during the development and release of a product.

[14] DoD CIO Enterprise DevSecOps Fundamentals, Version 2.0, March 2021

Recommended mitigations

The supplier and developer management team should set policies that ensure development organizations have security-focused principles and guidelines in place to:

- Generate architecture and design documents,
- Gather a trained, qualified, and trustworthy development team,
- Create threat models of the software product,
- Define and implement security test plans,
- Define release criteria and evaluate the product against it,
- Establish product support and vulnerability handling policies and procedures,
- Assess the developers' capabilities and understanding of the secure development process and assign training,
- Document and publish the security procedures and processes for each software release.

Architecture and design documents

Architecture and design documents should be based on customer and marketing requirements that have been gathered, correlated, and prioritized. Specific security-related assessments and reliability criteria derived from operational customer environments and known product risk assessments should be included in the requirements. The requirements should take into account security criteria for specific industries such as NIAP, FedRAMP, HIPAA, or FIPS-140 and that are based on Zero Trust principles. Architecture and design documents should address all requirements defined and describe the components, interfaces used, and functionality needed to implement the product in various levels of detail based on the needs of the development group.

The development team

Members of the development team should be trained and qualified to perform the security development tasks outlined in the architecture and high-level design document.

Threat models

Impartial, senior-level security architects and developers should create threat models of the product under development. These personnel should be familiar with identifying trust boundaries, relationships, and inflection points where data or systems might be compromised. Threat models should be developed for all critical software components, as well as for all critical systems in the build pipeline.

All code and systems involved within the build pipeline should be reviewed on an ongoing basis against the associated threat model. Changes should be made as needed to ensure neither the code nor systems have structural vulnerabilities. Threat models should further be:

- Updated as functionality changes, for major releases, or minimally at least annually,
- Made available to other internal engineering teams that are picking up or operating any associated software components or systems.

Management policies should also specify that developers assigned to create the threat models use component-level designs for completeness. Models should be reviewed and approved by at least two independent engineers on the team and evolve as architectural and design changes occur. The threat model process needs to be adaptive when organizational policies and procedures change.

Security test plans

An impartial Quality Assurance (QA) individual, team, or an impartial entity with QA expertise should define and implement security test plans.

A QA team is comprised of automation and build expert(s) who leverage modern techniques required to apply secure testing strategies for all components defined within the architecture and high-level design documents.

Developers should perform unit- and system-level security tests that are validated by QA. This allows QA to perform further security testing to cover a broader and deeper set of tests with less duplication of effort. The strategies defined within the test plan should include:

- Code coverage, which is integrated into each build and tracked as part of implementing the test and development plan,
- Baseline levels of code coverage should ideally be achieved on all code that is checked in, before new code is committed,
- Policies should be defined to maximize code coverage and address the SSDF tasks defined in PO.2.1, PW.5.2 and PW.8.2 of the National Institute of Standards and Technology (NIST) Special Publication (SP) 800-218 Standard,
- Test coverage should identify the percentage of code paths the test plan covers as well as the types of test tools used.

When release readiness criteria are defined, they may include requirements for the following types of tests:

- Static and dynamic application security testing (SAST and DAST) should be performed on all code prior to check-in and for each release using a standard set of company-approved tools. Results of testing should be documented, and all discovered vulnerabilities should be analyzed and addressed,
- Software Composition Analysis should be performed on all third-party software to include review against the MITRE Common Vulnerabilities and Exposures (CVE) and the NIST software security vulnerability bulletins. (NIST SSDF PW.3.2),
- Fuzzing should be performed on all software components during development to ensure that they exhibit expected behavior with different inputs. Results should be documented, and any anomalies or vulnerabilities should be addressed,
- Where possible, plan to employ memory-safe programming languages to mitigate a large portion of the most common exploitable vulnerabilities,
- For many types of software products including security software and general-purpose operating systems, many government customers may require independent lab testing

against a National Information Assurance Partnership (NIAP) Protection Profile (NIAP CCEVS),[15]

- Verification should be done to ensure that applicable anti-exploitation features are leveraged in development depending on the platform on which the software will operate. Such features complicate or prevent exploitation of many classes of unforeseen vulnerabilities. The Application Software Protection Profile v1.4[16] includes requirements for "Anti-Exploitation Capabilities" under FPT AEX Ext.1 and is available at niap-ccevs.org under "NIAP-Approved pps",
- Penetration testing should be done as routinely as possible, but not less than once per year, depending on potential risk (e.g., cloud products should be pen-tested more frequently),
- Use a testing approach that considers only externally visible behavior of the product without knowledge of the code, nor the inner working of the software to assure that repaired vulnerabilities are truly fixed against all possible compromises.

The results of **all** security testing should be documented, security defects should be fixed, and a synopsis of the test results should be made available to customers. This synopsis should include any Common Vulnerability Scoring System (CVSS) scores. The QA results should be used as one of the measurements of a product's readiness for release.

Release criteria

The management team should establish, manage, and apply release criteria and evaluate whether the product satisfies the criteria. The criteria should include:

- No unacceptable security vulnerabilities found when performing all required threat modeling and testing are pending,
- Cybersecurity hygiene of the development environment was maintained during development, as described in **Section 2. Developer**, and the relevant artifacts were collected and securely stored for future reference,
- Products were developed following the secure software development practices and tasks set by the organization, and relevant artifacts were collected and securely stored for future references. Examples of the artifacts are the design and architecture documents (ex. system and software component data flow, UML model), the threat model, verification and test results, revision history of software design, all the components, and a list of open issues and known vulnerabilities,
- Produce, correlate, and validate a Software Bill of Materials (SBOM). Contents of the SBOM are described in Section **2.3.5 Threat scenario: software bill of materials (SBOM)** and the National Telecommunications and Information Administration (NTIA's) *The Minimum Elements for a Software Bill of Materials (SBOM),*[17]
- The product management team ensures that all released binaries are digitally signed with a key associated with a root certificate from a trusted certificate authority,

[15] https://www.niap-ccevs.org/Profile/PP.cfm
[16] https://www.niap-ccevs.org/Profile/info.cfm?ppid=462&id-462
[17] https://www.ntia.doc.gov/report/2021/minimum-elements-software-bill-materials-sbom

- All released software meets company-wide cryptographic standards. These standards should be based on relevant industry best practices or (for federal agencies) applicable government standards such as *NIST SP 800-175B; Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms* and be enforced with an appropriately defined responsible, accountable, consulted, and informed (RACI) matrix,
- All shipping of open source meets company-wide standards including vulnerability assessment of the source and this information is made available to development groups. Ship the latest stable versions of open source, removing or providing a support plan for any open source software that has reached end of life, and ensuring licensing, if any, is fully understood and compliant with the open source usage policy.

Product support and vulnerability handling policies

The management team defines the product support and vulnerability handling policies and procedures as they address the entire lifecycle of the product from conception to end of life (EOL).

- Using a vulnerability submission system, all known security issues and vulnerabilities should be collected and tracked as product defects in the organization's defect tracking tool. This includes common weakness enumeration (CWE) and CVSS scores, specific impacts on the component, and any other relevant supporting data. Vulnerability information should only be stored in access-controlled pages in the defect tracking system, given the potential sensitivity,
- The organization should have a central company-wide The Product Security Incident Response Team (PSIRT) that supports a public-facing reporting tool (for example a web page) that makes it easy for external researchers to report vulnerabilities in the organization's products. The PSIRT team should work with external researchers to acknowledge and gather information on any reported vulnerabilities, and to ensure that any reported vulnerability is fixed. Organizations should practice responsible disclosure on all vulnerabilities,
- Updates to all in-field software, including patches and product updates should be delivered using a secure protocol like HTTPS/TLS. The in-field software products should perform integrity or signature checks on all delivered files to ensure the files are valid. This applies to delivering updates to both on-premise and in-the-cloud software products.

Assessment and training

The management team defines policies and procedures used to assess developers' capabilities and understanding of the secure development process. These policies should address:

- Who requires training,
- How frequently they must train,
- Who is authorized to conduct the training,
- The training topics,
- How to evaluate the trainees ability to meet the standards established by the training.

Security training for the development team is ideally conducted by a centralized, expert security team who can help product teams grow their expertise in secure development. It also provides engineers a point of contact when they have specific security questions.

The training should include:

- Secure software development and design,
- Secure code reviews,
- Software verification testing,
- Use of security and vulnerability assessment tools during development.

Developers should take regular and relevant security training, both for common topics and those deemed necessary for the individual role. Successful completion should be tracked for all engineers. Organizations should ensure individuals complete security training commensurate with the impact level of the system and software to which the individuals are assigned.

Engineers within the development organization should also be required to take annual training of organization-approved cybersecurity best practices. An example of this training would include how to spot suspicious emails and the point of contact for reporting a suspected breach. This training should include a test at the end of the course to ensure understanding of the material. See also NIST SP 800-50, "Building an Information Technology Security Awareness and Training Program," for more information on how training should be conducted and measured.

Individuals within a development team should be evaluated periodically, at least annually, to measure their knowledge and compliance against product security goals. At a minimum, this should be a representative survey, displaying a team's or individual's awareness of required corporate training, and any artifacts that attest to compliance with policy. Gaps should be examined to determine and address root causes, e.g., if there is a lack of usable tools to implement organizational security expectations.

Security procedures and processes

The management team documents the security procedures and processes. These documents should be reviewed, updated, and to the extent possible, made publicly available for each software release. This must be done without divulging sensitive security information about the product. These are living documents, which are reviewed both when questions arise during the development of the product and after the product has been released in a formal "after-actions" report or "lessons-learned" session with all members involved in the secure development process.

Alignment with SSDF

The mitigations provided in this section align with the activities found in NIST SP 800-218, "Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities." The following table aligns tangible development activities with the SSDF recommendations:

Table 1: Mitigation alignment with SSDF

Mitigation		Activity in SSDF v.1
Architecture and design documents		PO.1.1, PO.4.1, PO.4.2, PW.4.3, PW.3.1
Development team trained in secure development		PO.2.2, PW.1.1, PS.1.1, PS.3.1, PW.4.2, PW.4.3, PW.5.1, PW.5.2, PW.6.1, PW.6.2, PW.7.1, PW.7.2
Threat models		PW.1.1
Security test plans		PO.3.1, PO.3.2, PO.3.3, PO.4.1, PO.4.2, PW.4.3, PW.5.1, PW.5.2, PW.6.1, PW.6.2, PW.7.1, PW.7.2, PW.7.2, PW.8.1, PW.8.2, PW.9.1, PW.9.2, RV.1.1, RV.1.2, RV.3.3, RV.3.4
	Document results with CVSS scores; verify security defects are fixed	RV.3.2, RV.3.4, PS.2.1, PS.2.2
Product release	Deliver testing and threat model documentation, vulnerability reports, and SBOM.	PS.2.1, PS.2.2, PW.2.1, RV.1.2
	Support channel to report flaws.	RV.1.1
	Digitally sign shipping binaries with key and trusted root certificate	PS.2.1
Product support	Track known security issues/vulnerabilities	RV.1.1, RV.1.2, RV.1.3
	Incident response with public-facing reporting tools, fix reported items and disclose	RV.1.3, RV.2.1, RV.2.2, RV3.1, RV.3.3
	Update in-field products	PS.3.2
Assessment and training of developers		RV.3.4
Security procedures and processes for each release		RV.2.2
Cryptographic and third-party software integration standards		PW.3.2, PW.4.1

Note SSDF Activity Codes: PO – Prepare Organization; PW - Produce Well-Secured Software; PS – Protect Software; and RV – Respond to Vulnerabilities.

2.2 Develop Secure Code

Source code development involves reviewing the approved product requirements and design documents and implementing all required features and functionality. This should be done according to the policy and procedures for writing source code and in a specified computer programming language (e.g., C++, Java, Python, RUST, etc.) as specified in SSDF PO.1.1, PO.2.2, and PW.1 of *NIST SP 800-218*.

Care should be taken when there is an opportunity to select the programing language to be used for development, considering whether the language is statically or dynamically typed, and what

protections are inherently built into it to mitigate vulnerabilities and provide memory and thread safe operations. Secure software development follows the principles outlined by Saltzer and Schroeder in "The Protection of Information in Computer Systems,[18]" which include:

- Open design,
- Fail-safe defaults,
- Least privilege,
- Economy of mechanism,
- Separation of privileges,
- Total mediation,
- Least Common mechanism,
- Psychological acceptability,

Developers may also integrate common core libraries and reuse trusted modules which have already been vetted by the organization as defined in SSDF PW.4. In many cases these guidelines outline the approved security settings for compilers and the deployment of standardized development environments and tools as specified in SSDF PO.3 and PW.6. Source code will typically be version controlled and managed in a source code control system following the guidelines in SSDF PS.1.1 and PS.3, and developers may be required to perform peer-reviews of their source prior to allowing code to enter a main repository as specified in SSDF PS.1.1 and PW.7. At times, engineers are required to compare and merge changes across code lines and repositories to manage source code properly in a distributed team model.

2.2.1 Modification or Exploitation of Source Code by Insiders

The Cybersecurity and Infrastructure Security Agency (CISA) defines insiders as "any person who has or had authorized access to or knowledge of an organization's resources, including personnel, facilities, information, equipment, networks, and systems."[19]

CISA defines insider threat as "the potential for an insider to use their authorized access or understanding of an organization to harm that organization." This includes intentional as well as unintentional acts.

Software development group managers should ensure that the development process prevents the intentional and unintentional injection of malicious code or design flaws into production code. Source code modifications can occur at the developer level in one or more of the following scenarios:

- When an engineer is compromised by outside influence or dissatisfaction,
- When an engineer is poorly trained,
- When engineers put backdoors into a product,
- When remote development systems are not secured or when protections are removed,
- When accounts and credentials for terminated or inactive personnel remain available.

[18] http://web.mit.edu/Saltzer/www/publications/protection/index.html
[19] https://www.cisa.gov/defining-insider-threats

1. Compromised engineers

The compromised engineer is a difficult threat to detect and assess. A compromised employee may be under pressure from outside influences or may have a grudge to avenge. . Poor performance reviews, lack of promotion, or disciplinary actions are only a few of the events that might cause a developer to take action against an organization and sabotage its development effort. Additionally, nation states or competitors can leverage an insider's struggles with controlled substances, failing relationships, or debt, among other things.

Because a developer has inside knowledge of the code base and is often an expert in their respective coding language, environment, and style, developers can design subtle vulnerabilities that are very difficult to detect. In addition, access to design details not publicly available can provide inside knowledge of weak architecture or code areas that contain security weaknesses that may be exploited.

Once implemented and injected into the build infrastructure, built-in vulnerabilities are compiled, signed, and hashed, allowing the high-level security validation checks to pass without any indication of compromise.

2. Poorly trained engineers

Engineers who have not been properly trained in security design and coding practices can unintentionally introduce vulnerabilities within source code that, once submitted into a source control repository, can be difficult to detect. The type of vulnerabilities can range from buffer overflows to logic flaws, the latter being harder to discover. These "zero-day bugs" can reside in a product for a long time and are instrumental in providing an easy compromise vector for adversaries that discover them.

3. Ease of development features (backdoors)

Developers will sometimes add debugging features within a product to facilitate the troubleshooting, setup, or problem-reporting processes commonly performed before initial development. These features, in many cases, are privileged operations that allow the development team to obtain statistics and logs and issue remote commands to reconfigure the system under development. While these developer features can be helpful, often they are tightly integrated into the product's core components, making them hard to remove. In some cases, they cause components to be "extended" to facilitate the tools and features being used but not formally designed within the product.

These features are often planned to be completely removed before release, but in some situations, they are not removed due to the core component integration and the level of work or risk involved with removing them near the scheduled time of product delivery. These features could be disabled upon release, but when left in the shipped product they create risk of discovery and exploitation. Another ease of development concern is when only one portion or function of an application requires elevated privileges, but the entire application is configured to run with the privileges required to perform the single task. Privileges should be raised to complete specific functions and then immediately lowered to reduce the attack surface.

4. Compromised remote development systems

A common practice within the development environment is allowing remote development for employees or contracting off-site third-party developers. Many of these remote developers work from home or a satellite office and use organization-supplied machines and resources connected over a VPN. When using this environment, the remote office becomes an extension of the organization's network, and the developer has access to all the development resources normally associated with a standard work environment to include creating, compiling, and checking in source changes.

While there are many benefits to facilitating this work environment, remote work comes with risk. The home or remote office network may not provide the same level of network protection as a company's on-premise facility. In addition, remote employees may be more tempted to use restricted network-based applications for social media, web surfing, games, and in some cases removing local computer protections to facilitate their use. In this environment, these systems can become compromised, allowing an adversary to use backdoors within the remote environment to access and modify source code within an otherwise protected organization infrastructure.

5. Use of lingering accounts or credentials of a terminated or inactive user

When employees have been terminated, reassigned to another project, or away from work for an extended duration, their privileges and accounts often remain operational, and may be used to perform malicious activity without the account owner's knowledge. In this case, the owner of the account is not monitoring its use and is unaware of any malicious activity performed with it. Unauthorized use of accounts in this way grants access to all the development resources available to the original account owner.

Recommended Mitigations

Specific processes may help mitigate the risk of intentional or unintentional injection of malicious code in a development project, including:

- Implementing a well-balanced authenticated source code check-in process,
- Performing automatic static and dynamic security/vulnerability scanning,
- Conducting nightly builds with security and regression tests,
- Map features to requirements,
- Prioritize code reviews and review critical code,
- Secure Software Development/Programming Training,
- Harden the Development Environment.

1. Implement a well-balanced authenticated source control check-in process

Fundamental to the protection of the source code repository and its contents are the methods used to control access to it and the validation process used to ascertain whether a check-in is "good." Access and validation start with good source code management (SCM) principles to track modifications to a source code repository. Such principles include a running history of changes to a code base and resolving conflicts when merging updates from multiple contributors. As an example, the acquisition processes for free and open source software,

commercial off the shelf software source components, and the management of a secure software repository are outlined in Figure 3.

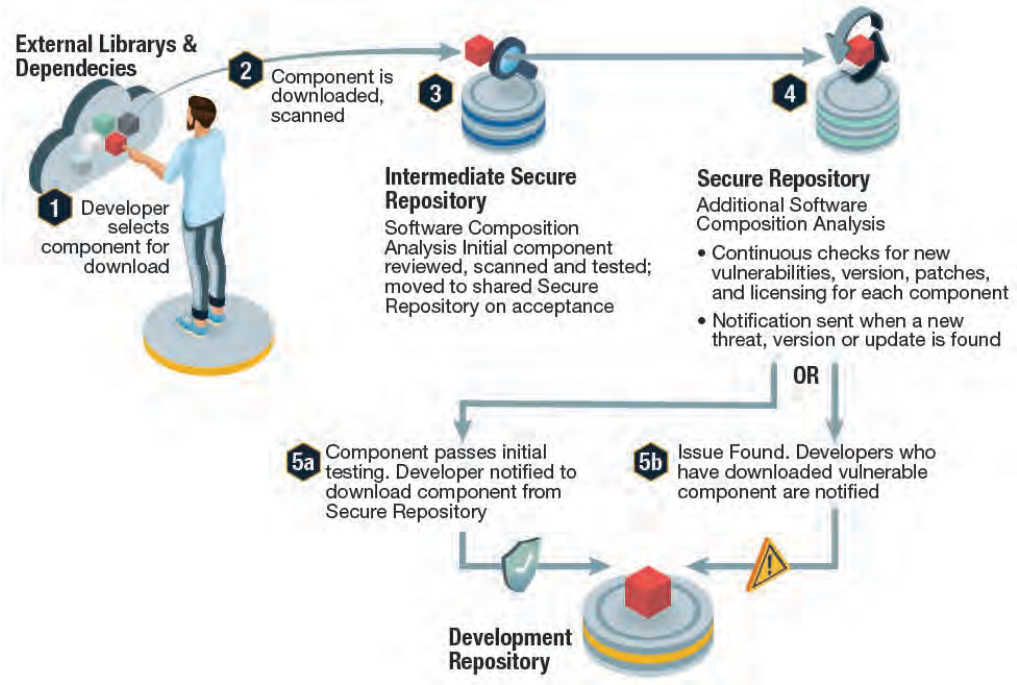

Figure 3: Secure Repository Process Flow

The secure repository should initially and continuously look for new vulnerabilities and updates within the added components. A log of all developers and the components they download should be kept. If a component becomes flagged due to a new vulnerability or update in the future, the developers who have downloaded the component should be automatically notified to address the issue. In this manner, when new vulnerabilities arise, it will also be evident which programs/projects are affected.

At a minimum, the source control system should be protected using industry recognized multifactor authentication (MFA), not only to log check-ins, but for all access to the secured repositories. When check-ins are made, an audit trail is created that logs the MFA developer ID, files modified, and date and time of the check-in. Depending on the complexity, security requirements, development resources available, and time constraints, consider the following when implementing a well-balanced source control check-in process:

Peer/lead review

- Allow no code that has not been peer or lead reviewed to be checked-in to a source control repository,
- Require comments listing the relevant requirement for the check-in,
- Include the MFA ID of the reviewer and the reason for the modification of the source,
- Include any cross-development dependencies on another development effort co-dependencies should never be checked-in separately,

- Perform unit and security tests.

Working and production branches

To control the quality of the produced software, two or more branches of the development tree are maintained. During the normal software development process, all code can be stored in the working, general purpose, development branch. As a component development effort evolves, the source supporting a delivery feature is coded, tested, and reviewed by senior engineers, and the functions and requirements of the component are cross-referenced. This ensures the feature set is met and nothing exists for any feature creep. Approved code is moved to a production branch by a development integrity assurance team made up of senior level engineers, build engineers and designers. The production branch, sometimes referred to as the release branch, is used as the sole repository from which release product is built. This branch should be protected with reviewers and continuous integration/continuous delivery (CI/CD) tests with SAST enforced at the SCM. The process flow for branch readiness and transfer could be summarized as:

1. Developers work in the development branch.
2. Leads promote software to a QA branch when source is code reviewed and approved.
3. QA individuals/teams test from QA branch.
4. Once integrated code is tested and approved it is moved into the production branch.

Access to the production branches is restricted to a small set of build and development team members. All builds used to create production products are created from the production branch of the repository. Once a product is released, the product branch should be labelled and locked down with restricted customer or read-only access. The implementation of this lockdown procedure ensures secure and reproducible builds.

2. Perform static and dynamic security/vulnerability scanning

Performing automatic static and dynamic vulnerability scanning on all components of the system for each committed change is key to the security and integrity of the build process and to validate the readiness of product release. This automatic scanning can perform code analysis to determine if restricted application programming interfaces (APIs) that contain vulnerabilities such as a buffer-overflow or memory leakage are used within the source under evaluation, as well as other security related scanning. Performing static analysis to scan for secrets before commit and during CI/CD blocks secrets from the code base. The complexity and thoroughness of these static scanning technologies vary greatly. These tools should be used by test teams as well as locally used by the development engineer. Most secure development processes recommend this practice.

Separate and higher quality scanning tools should also be used within the product build environment. It should also be noted that static analyzers work better on statically-typed languages such as C++, since the type of variables used within the code are known at compile time, whereas dynamically-typed languages such as Python resolve the variable types at runtime. As functions and components are completed and able to be executed, dynamic testing can find additional security weaknesses. These are often user input errors or malicious injections and can only be identified during testing at runtime. For web applications, Interactive

Application Security Testing (IAST), Dynamic Application Security Testing (DAST) and Runtime Application Self-Protection (RASP) tools should be used, as specified in NIST 800-53 v5.[20] Because IAST tools tend to have far more false positives than SAST, particularly with web applications, SAST tools that use introspection are encouraged when implementing the security testing requirements within this environment.

3. Conduct nightly builds with security and regression tests

To ensure the integrity and quality of the development process, nightly builds should be performed that include manual and automated security and regression tests. Test cases should be implemented during the design of the software and extended during coding to validate all areas of functionality for both "good" and "bad" scenarios. Using this process, any flaws or changes, whether malicious or inadvertent, can be recognized and addressed.

The nightly builds with regression tests should be implemented by a QA engineer and incorporated into the build environment by a build engineer. This is different than the case of a developer's own automated unit test, which may run manually during coding and automatically during the "building" of the component and for which the developer is generally responsible.

Artifacts such as logs and automated email notifications sent when regression tests fail help notify and track where and when problems arise. The nightly build process also acts as a good performance matrix to assess the developer capabilities and comply with security and development processes. Refer to Section **2.4 Harden the Build Environment** for more details related to production builds of the product as compared to local development builds documented in this section.

4. Map features to requirements

It is important that all components and functionality of a product are architected and designed to interact with the system using secure design practices, including threat modeling and attack surface analysis. Once all security risks are identified and mitigated, architecture and design documents are finalized and disseminated to development groups for implementation. Low-level design and functional specifications are created that map directly to the given architecture and high-level design, and development tasks and schedules are mapped out. During the coding and implementation of the system, care must be taken to ensure that all development efforts map to specific system requirements and that there is no "feature creep" that might compromise product integrity or inject vulnerabilities.

Formal, informal, and peer reviews help ensure that code added to the repository meets specific requirements and only those requirements. These reviews can also identify modules and unused code that are included as part of a larger package or component feature. When possible, only required modules should be included in the product.

Additionally, developers should remove unused modules and code that is out of scope of the requirements and design document. Restricting the addition of developer tools, like those that aid debugging, configuration, and monitoring, to only those approved within the design of the

[20] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf

system also mitigates the attack surface. Reporting and addressing feature creep as soon as possible helps as well.

In addition, the build environment should support the scanning and detection of all plug-ins within the system. The results should be cross-correlated with an allowed list to ensure unauthorized components haven't been added. These scans produce artifacts that describe which components and software features are included in a product, and, more importantly, ensures that all components have undergone analysis so each component's risk is understood and documented.

5. Prioritize code reviews and review critical code

Code reviews are performed using two different processes, formal and informal, and are implemented during different stages of the development lifecycle.

Code reviews should be prioritized, and, at a minimum, the most critical code should be identified and reviewed using both the formal and informal review processes. Critical code includes components that:

- Use or provide cryptography,
- Require privilege escalation,
- Access protected resources,
- Are essential to the purpose of the software, or
- Have a high percentage path flow, among other factors.

In addition to formal and informal code reviews, automated code review tools should be deployed to provide full code review coverage.

The informal code review process is used by the internal development group when measurable stages or checkpoints are achieved during development. The informal review is used to ensure secure coding policies and procedures are being followed and that the source under review meets the design and functional requirements documented within the low-level design of the component. These reviews are conducted by members of the development team to include project leaders and senior developers and ensure security and integrity during development of the specified component. While informal, the process for conducting code reviews produces documentation on the areas reviewed and the results, which can be used to help measure a development group's performance. Informal code reviews are also used as a training tool to provide awareness to group members on how to implement secure coding practices. Furthermore, informal code reviews can be used to measure the quality of the code review process and developer deliverables being produced.

Formal code reviews ensure the secure integration of key components using best practices. These reviews focus on the integrity of a system, design, and architecture, while addressing all functional, security, and reliability concerns. They also identify any areas of concern that may be exploited or compromised.

Engineering groups, including members of QA, build experts, designers, and architects, usually conduct formal code reviews. In formal code reviews, before review of the source, the owners of the component describe the component's functionality and its interaction with other

components in the system. A designated individual takes detailed notes that track the review of core components of the product.

The outcome of the formal code review is a list of activities and concerns that must be addressed prior to shipping the product. Formal code reviews aid the development effort by sharing knowledge of the product design and implementation which enhance the build and testing requirements of the development environment. Formal code reviews also allow the product management team to measure the product's readiness for release, ensuring it meets all requirements prior to release.

6. Secure Software Development/Programming Training

As outlined in **Section 2.1, "Recommended Mitigations,"** subsection "**Assessment and training**," developers should continually take training to understand the secure development practices required for corporate development. This resource must also be made available during the development lifecycle to allow developers to contact experts and address concerns or questions they may have about a specific security issue, practice, or vulnerability. During day-to-day development activity, access to security experts should be used to facilitate the peer and source code reviews and to train individuals on corporate security practices and evaluate their knowledge. A well-implemented peer review process allows engineers to prevent defects, minimize security weaknesses, and promote team collaboration and knowledge-sharing.

7. Harden the Development Environment

As with the build environment, the development environment must be hardened. The mitigations defined in Section **2.4 Harden the Build Environment** also apply to the development environment. However, while the production build systems are generally located in a protected segmented network with limited access, the development build environment may be housed in endpoint systems that are less isolated. For example, many development environments allow remote VPN access to the internal development machines to facilitate remote workers and allow them to participate in development activities. In such situations, when connecting to a corporate development environment from a remote location, a VPN and MFA must be used to protect the development environment. Endpoint security software should be installed to prevent, detect, and respond to threats against the host system. In addition to VPNs, implementers should consider the use of a "jump-host," a system which acts as a portal between a less secured remote host and the protected development environment. This allows all activity to be continuously monitored and providing protection and/or limited accessibility and operational privileges for the remote developer. A threat model and vulnerability assessment are required for all development environments associated with product development.

2.2.2 Open Source Management Practices

Developers commonly use open source code in application development, with projects potentially having multiple dependencies on open source libraries which may contain vulnerabilities.

Recommended mitigations

Development organizations should employ dedicated systems that download, scan, and perform recurring checks of open source libraries for new versions, updates, and known or new vulnerabilities (see Figure 3 above). As with all software, we strongly recommend educating developers on considerations for the use of open source software, close-source software, and evolving best-practice mitigations. Please refer to the SSDF for more details, specifically PW.4.1, PW.4.4, PW.4.5 and PO.1.3.

2.2.3 Secure Development Practices

Managers of a software development group should ensure that the development process used to generate, test, and preserve source code are accomplished using well-defined and secure practices. This helps establish trust in the engineering tools-chain and procedures used. These practices address the following security concerns:

- Secure developer environment,
- Use secure development build configurations,
- Use secure third-party software tool-chain and compatibility libraries.

Recommended mitigations

The following are recommended activities to implement secure development practices:

1. **Secure the developer environment**

   When ensuring the integrity of the development environment, care must be taken to harden the development systems within the build pipeline as defined in Section **2.4 Harden the Build Environment**. In addition, all development systems must be restricted to development operations only. No other activity such as email should be conducted for business nor personal use. If possible, development systems should not have access to the Internet and may be deployed as local virtual systems with host-only access. All tools installed on development systems must be pre-approved, to include debuggers, test tools, vulnerability scanners, and modeling software even when confined to single local development use.

2. **Use secure development build configurations**

   Many exploits use common compromise techniques such as buffer overflows, return-oriented programming (ROP) execution gadgets, delayed dynamic function loading, and overriding Software Exception Handlers (SEH). For many of these techniques, compiler, assembler, linker, and interpreter tools have been extended to include defenses to mitigate these risks. The following is a list of build-chain defensive techniques that should be deployed to narrow the compromise vectors:

   a) Stack Cookies – Prevents stack overwrites,

   b) Address Space Layout Randomization (ASLR) – Prevents ROP/Hardcoded IP references,

   c) SEHOP – Prevents SEH hooking,

   d) Data Execution Protection (DEP) – Stack/Heap execution prevention,

e) No Execute Bit (NX) – CPU flag execution prevention of memory locations,

f) Static Libraries – Prevents preloading of malicious dynamic libraries,

g) Stripping Binaries – Removing symbols from binary files makes it harder for the file to be reverse engineered,

h) Hardware Specific Preventions – More preventions are available based on built-in hardware support.

While the above preventions are crucial for ensuring the runtime protection of software under development, the development team should also provide to the end user a suggested environment for which their software may run securely. For example, many antivirus products provide behavior analysis engines to provide additional security checks, for example:

a) Heap Spray Mitigation – Monitoring commonly targeted heap addresses,

b) Stack Pivot Detection – Detects ROP,

c) ROP Call Detection – Detects JMP/RET (unconventional program flows),

d) DLL Injection Detection – Dynamic Link Library (DLL) location and signature validation,

e) Null Page Detection – Dereference exploitation prevention,

f) Root-Kit Detection – Address hooking prevention,

g) Behavioral Heuristics – Detection of unusual CPU, memory and resource activity.

It is also important to note that while interpretive languages generally do not have the vulnerability risks outlined above, the implementation of the interpreter itself and the underlying system libraries they use do, therefore these mitigations hold true for them as well.

3. Use secure third-party software toolchains and compatibility libraries

In developers' build processes, various tasks are often integrated within an Integrated Development Environment (IDE). Many of these environments are self-contained and allow all development, to include coding, compiling, linking, packaging, and debugging, to be performed from within the tool. The IDE may even provide the ability to check-in a source to a repository. In many cases, these IDEs support multiple compiler languages and environments and the ability to extend the IDE by installing plug-ins. Because of the complexity and untrusted sources, IDEs may become compromised, leading to an insecure local development environment. To ensure the integrity of the development process, all IDEs and their associated plug-ins used within a developer environment must be preapproved, validated, and scanned for vulnerabilities before being incorporated onto any developer machine.

Build environments may require the use of operating system specific utilities and commands. For example, a Windows environment may require Linux operating system commands during the build process on the developer host. Such build environments may necessitate the need for third-party operating system tools and utilities to be installed on the development host to provide compatibility between the development environment and the production build environment. In addition, many development environments require API conversion libraries to

facilitate a common coding environment between two disparate operating systems such as Windows and Linux. Toolchains and compatibilities libraries are available through commercial and open source software. Both the compatibility toolchains and libraries also need to go through a vulnerability assessment prior to being adopted within the development environment.

2.2.4 Code Integration

Development managers want to ensure that the components and software integrated into the delivered product is tested within the integrated environment for which it will be deployed. This process involves incorporating all required dependencies including source code, components, and additional required metadata and utilities into a single system. During code integration, the developer will ensure that the code can be successfully built, and monitor and evaluate the runtime behavior. Software should be integrated using zero trust principles as recommended in NIST SP800-207.

Recommended mitigations

All third-party modules should be tested for known vulnerabilities against the Common Vulnerabilities and Exposures (CVEs) that are listed in the National Vulnerability Database (NVD). A software composition analysis tool can help automate this process. The development team should also subscribe to alerts and reports from the National Cyber Awareness System and other sources for the latest software vulnerability information. Once modules are reviewed for vulnerabilities, they can be added to a developer or Open source Review Board (OSRB) repository for all approved downloaded modules. This trusted repository should continue ongoing testing to identify new vulnerabilities that are reported within the modules. It should also incorporate a process to provide vulnerability updates and/or patches to the end-user. Applications include code from other sources, sometimes slightly modifying or adding integration code for their specific use purpose.

There are security dependency analyzers and many other tools and services that can help detect reused components with known vulnerabilities. These activities are typically conducted in an Integrated Development Environment (IDE) and use the organizations' secure coding practice and guidelines such as in PW.3.1, PW.3.2 and PW.4.1 of the SSDF. Code integration should be implemented using zero trust principles. Trust should not be implied and therefore critical components and functions should check usage and access rights within the code and only use escalated privileges when necessary. Developers should ensure that code and build integration process is repeatable. Developers and QA engineers may provide automated regression tests to ensure components are integrated properly and functioning as specified in the design and requirements document. They may also provide additional static and dynamic scanning tools to detect coding errors and security flaws within the developed code as defined in SSDF PW.5.1 and 5.2.

2.2.5 Defect/Vulnerability Customer Reported Issue

Managers of a software development group should ensure that the software they develop is free of high-risk known defects and vulnerabilities. When vulnerabilities are discovered and reported by the customer, the development group should respond to the incident and provide component updates to mitigate the risk associated with the defect or vulnerability.

Recommended mitigations

Suppliers should have a public process to accept reports of potential defects and vulnerabilities from customers and third-party researchers. Suppliers should use automated vulnerability notifications from trusted organizations such as the Cybersecurity & Infrastructure Security Agency (CISA) to receive timely alerts of the recent and high-risk threats. All notifications should be evaluated with respect to the relationship to the product and prioritized based on risk assessment.

Engineers should then be assigned to review, diagnose, and resolve issues as defined in PW.8, RV.1.1 and RV.1.3. To decrease the attack surface, a process should be implemented to identify the class of the vulnerability and examine other product features and components that might potentially be affected by the same identified class of incident. Customers should be provided timely responses with the organization's internal vulnerability management policy, which should be based on industry best practice documents such as NIST SP 800-216 guidelines. Updates to software are made available using a secure channel as required and specified in SSDF RV.2.1, RV.2.2. The update is also made available and communicated to all customers of the product describing the defect or vulnerability and resolution. Updates may also be automatically applied to a product based on the update strategy configured by the customer.

2.2.6 External Development Extensions

Once released, product functionality may be extended by a development team other than the original product development team. In many cases, this external development team, with respect to the development responsible for the product, may need to add additional functionality to the product or customize it for specific customers' needs which were not met or implemented by the owner of the product. This external development team may be a solution team within the company that produced the product or a Value-Added Reseller (VAR). An example of the type of activity that might be performed is the addition of a required authentication method to the existing product.

When such an activity occurs, modifications to the product can include the addition of software packages to support the feature, as well as graphical user interface (GUI) changes to enable and manage the new functionality. During this activity, vulnerabilities may be introduced by the new development, the new packages deployed, or modifying a provided API that is not being used as designed or intended.

Recommended Mitigations

When possible, extensions to a product must follow all secure development practices as the originating product development as defined in SSDF PS.1.1, PW.4.1, PW.4.2, PW.5 and PW.7. In addition, a Software Bill of Materials (SBOM) should be made available that details the additional packages and software that was added. If signing is required, the certificate that is used (if not from the same supplier of the product) must be clearly identified. Modules that are modified from original source must be clearly identified within the new SBOM and original component information and owners identified along with all the new information required to describe the modified module, as required by SSDF PS.1.1.

The PSIRT must be available and ready to assist end users when problems occur, even if the cause of the problem is related to the extensions added as required in SSDF RV.2.1. In many cases, the VAR will act as a "go-between" between the customer and the product PSIRT to help resolve issues.

Feedback to engineers to resolve issues must be aggregated between the customer and external development group to allow timely, accurate information which may be used to resolve issues. Any code modifications due to vulnerabilities within the provided APIs and functionality must be corrected and published to all customers. The activities provide mitigations in line with the following SSDF activities:

1. PS.1.1 - Store Code and Executables, and Review and Approve All Changes.
2. PS.3.2 - Create and maintain a SBOM for each software package created.
3. PW.4.1 - Acquire well-secure components.
4. PW.4.2 - Create secure software components in-house.
5. PW.5 - Create Source Code Adhering to Secure Coding Practices.
6. PW.7 - Review and/or Analyze Human-Readable Code.
7. RV.2.1 - Analyze each vulnerability to gather sufficient information to plan its remediation.

2.3 Verify Third-Party Components

Developers routinely incorporate third-party commercial software components as an aspect of their activities to leverage existing Application Programming Interface (API) capabilities. These components may be Free Open Source Software (FOSS) or Commercial Off-the-Shelf Software (COTS). When sourcing these components, developers will typically make their selection based on criteria including the capabilities the component enables and the sustaining engineering support model for the component. Prior to the incorporation of third-party components by engineers, an organization may require an approval process as outlined in Section **2.2.4, Code Integration.** This process may include vulnerability database analysis, secure composition analysis, vulnerability analysis, risk assessment, and source code evaluation on the components under consideration, the results of which indicate whether the specific components identified are allowed or not. Once selected, the identified components are continually monitored, if possible, by using an automatic vulnerability tracking service that prioritizes and fixes identified vulnerabilities within open source components. PSIRT teams may discover new vulnerabilities and alert product owners to remediate when a new vulnerability is discovered.

2.3.1 Third-Party Binaries

Third-party software, sometimes delivered in binary format, is like a black box for the engineer or the organization who is integrating it. The software may not be actively maintained and may have security weakness or vulnerabilities.

Recommended mitigations

1. Binary scanning and software composition analysis tools can often detect unknown files and the open source components contained in binary packages, identifying the security weaknesses associated with these components without the need for source code. The tools may evaluate and provide a score of the vulnerabilities detected. The activities are highly recommended to verify the integrity of the third-party software. The output can be compared with the SBOM, or the source codes provided by the third party, to verify the SBOM.

2. The development team runs the binary scan of the third-party software, identifies potential threats including unknown components, open software components and vulnerabilities. The output of the composition analysis should be considered in the organization's decision to select and integrate the software component. Please see Section **2.3.2 Selections and Integration** and **2.3.3 Obtain Components from a Known and Trusted Supplier** for more information.

2.3.2 Selections and Integration

Before the integration of third-party components, each component must be evaluated for the potential security risk that might be associated with it. The evaluation includes reviewing and testing the software.

Recommended Mitigations

SAST/DAST and other appropriate review such as composition analysis must be performed to determine if the risk is acceptable. Once determined, the source code (not binaries alone) should be integrated into the build environment allowing the security scanning processes of the build environment approved by the organization to take place. Whenever possible, images should be built from the source and not downloaded from the internet, unless there is an understanding of the provenance and trust of delivery.

2.3.3 Obtain Components from a Known and Trusted Supplier

When considering the selection of a third-party component, care should be taken to build a relationship with a known and trusted supplier that has a proven record for secure coding practices and quality delivery of their components.

Recommended mitigations

When the organization makes decisions concerning selection, use, changes, or updates of third-party or open-source software for its products, it should perform a risk assessment and ensure the residual risks are acceptable. The organization should verify a third-party's ownership and control status, their Data Universal Numbering System (DUNS), and past performance of the suppliers and their upstream suppliers, if such information is available,. The selection will also take into account the producer's country of origin and adhere to the Defense Federal Acquisition Regulation Supplement (DFARS), as required.

Suppliers should produce artifacts attesting to the development process, quality, and security aspects of the component being considered for inclusion in an organization's software product. The availability of artifacts does not exclude the process listed in Section **2.3.2 Selections and Integration**. In addition, an organization will compile a list of known trusted suppliers and their associated artifacts that have been integrated into the company's products as well as a repository of the third-party components that have been vetted. A record of all components that comprise a product are captured within an SBOM to implicitly aid in the verification and vetting of the product in its entirety. Trusted suppliers are measured by:

1. The third-party component meets all requirements for the product considering adoption.

2. The quality of the third-party component is verified based on the results of testing by the adopting organization.
3. The quality of the artifacts supplied, for example the validation of an SBOM, is validated as being correct.
4. The owner of the component has a history of timely responses to known vulnerabilities reported within the third-party component.
5. The third-party mechanism to report vulnerabilities is easy to use. All available updates of adopted components are easy to integrate into the development environment using well defined and understood update procedures between the third-party and development group.

2.3.4 Component Maintenance

Once a third-party component has been selected and integrated into the product, care must be taken to monitor modification of the component by the supplier, specifically with respect to addressing know CVEs and vulnerabilities that have been reported by the development team and community of customers of that component. Adoption of changes follows the same process as outlined in Section **2.3.2, Selections and Integration**.

Recommended mitigations

The adopting product organization should monitor available CVE reporting mechanisms and third-party support channels to determine whether vulnerabilities identified within an adopted third-party component can impact the products and take appropriate actions to solve or mitigate the vulnerability. The contract with the third party should resolve future vulnerabilities. The owner of the third-party component must also notify the product organization of the presence of a vulnerability, the risk associated with it and a timeframe for when the vulnerability will be addressed and made available to the product organization.

2.3.5 Software Bill of Materials (SBOM)

The details of an integrated third-party component should be reported in an SBOM for the product developed to easily validate approved components and identify the presence of vulnerable components when defects are discovered. Several specifications define the format of an SBOM:

1. The Linux Foundation Projects “Software Package Data eXchange (SPDX).”
2. OWASP “CycloneDX.”
3. NIST “Software Identification (SWID) tags.”

Recommended Mitigations

An SBOM provided by a supplier or owner of the third-party component should be validated and updated as needed. Any discrepancies should be reported to the supplier. To that end, software composition analysis (SCA) tools should be applied to the software deliverable from the third-party. The third party's SBOM can be compared with the SBOM produced by the SCA tools. As described in **2.5 Deliver Code**, the binary scanning SCA tool can identify the contents of the final deliverables from the third-party software.

If an SBOM is not available from the supplier, the development team will derive the required information to describe the third-party component within the SBOM for example by utilizing software composition analysis tools. When third-party source is modified by a developer, both the initial SBOM, if provided by the supplier and the updated SBOM, describing the enhancement or defects addressed in the original supplied source, can be defined within the SBOM's dependency relationship element.

Please refer to Section 5 (recommended data fields) in NTIA's *The Minimum Elements for a Software Bill of Materials (SBOM)*.

2.4 Harden the Build Environment

This document outlines two types of build environments, the individual developer environment and the production build environment. An example of both environments are outlined in Figure 4. For more information on the individual developer environment, refer to section **2.2.3 Secure Development Practices**.

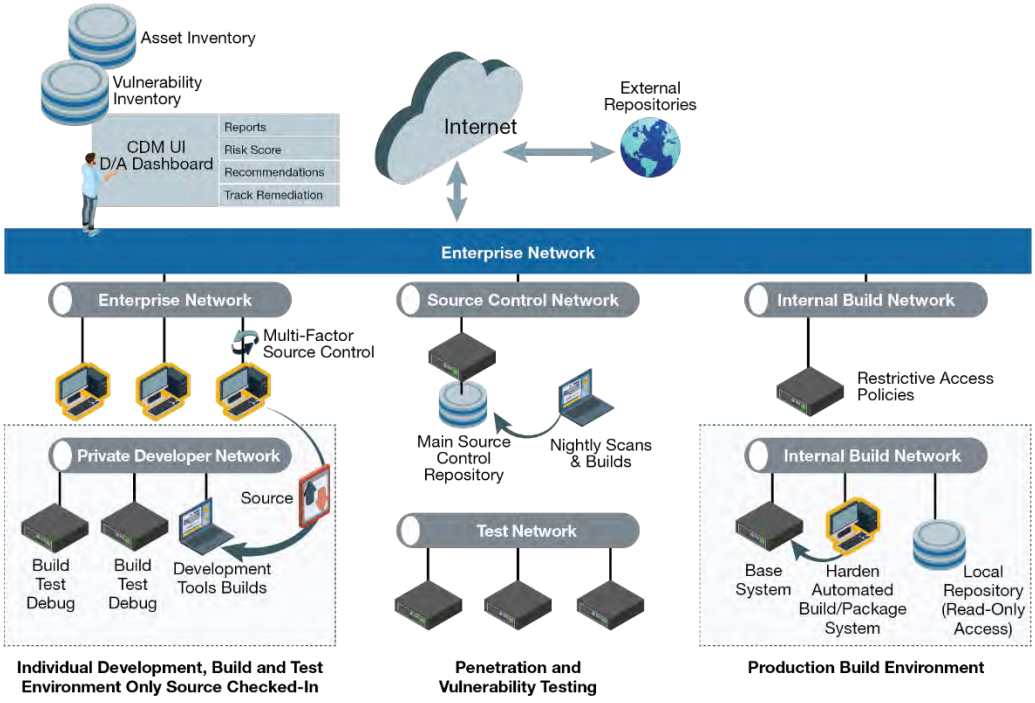

Figure 4: Secure Build Environment

The production build environment is where reproducible deliverables are built. The components that comprise a product are provided to end users as a bundle that may include multiple modules and a cryptographic signature. The cryptographic signature validates that the software has not been tampered with and was authored by the software supplier. The build process may include automated tasks to validate the security of the software. The software is installed by customers and, after some validation, put into production. The build environment may produce software that is

then deployed as software-as-a-service (SaaS). These applications provide some functionality over the network. The resulting software is usually not distributed to customers for installation. Other common build environments under consideration include the following:

- Continuous integration/continuous deployment. In this case, the software is installed in a subset of the cloud for immediate feedback and A/B testing,
- Building software as part of a rapid iterative cycle, such as using an Agile Development method. The resulting software may be distributed to customers or may be used for testing without distribution,
- Building software as part of an open source project, where executables are not distributed as an output of the project. In this case, the software resulting from the build is intended to be a baseline for testing and to identify problems early in the development cycle.

Common to all scenarios are the tasks associated with architecting, implementing, and maintaining or optimizing the build process. Also common are provisioning and configuring equipment as build servers or VMs, networking, and configuring user permissions.

The build system must be developed and maintained with the same level of security, integrity, and diligence as the source code and resultant product itself, as described in Section **2.1 Recommended Mitigations**.

The build environment may be hosted in on-premise systems or may be hosted in a cloud. The same rigor and discipline for hardening an on-premise build environment should be used for the cloud-based build environment.

Note: *It might be advantageous to build software in both cloud and on-premise environments and compare the results. If the results do not match, there may be evidence of supply-chain tampering.*

2.4.1 Build Chain Exploits

The build environment is a prime target in a supply chain compromise. In this scenario, a compromise may occur when a threat actor:

1. Infiltrates the development network.
2. Performs a scan to locate the repository and build systems and to identify vulnerabilities.
3. Crafts an exploit to compromise the build system or repository (or both.
4. Deploys the exploit.

In this case, the exploit is subsequently included in:

- Compiled source code,
- Included libraries,
- Reintroduced when libraries revert to older third-party libraries with vulnerabilities,
- Resident memory, which gets embedded in source during compile time via a rootkit-the source is not modified in the repository,

- Network Man-in-the-Middle (MITM) attack, which modifies source when being pulled down to build system.

Recommended Mitigations

The build pipeline infrastructure includes all systems that come in contact with the development and build process such as source code repositories, engineering workstations, build systems, signing servers, and deployment servers for both on-premise machines and those hosted in the cloud. Each of these systems should be:

- Completely locked down,
- Protected from external and off-local area network (LAN) activity,
- Monitored for data leakage, particularly code repositories and engineering workstations,
- Configured to prevent infiltration and exfiltration.

Additionally:

- Subject build scripts and configuration files to the same code review process listed in Section **2.2.1 Modification or Exploitation of Source Code by Insiders**.
- Use version control for pipeline configurations,
- Ensure each system requires multi-factor authentication,
- Segregate the engineering network from the corporate network,
- Minimize and regularly audit service accounts,
- Log all access to the build pipeline,
- Protect any secrets associated with the build pipeline.

Lock systems down

When locking down systems, only those operations specific to each system's function should be allowed. For example, build systems should only perform operations necessary to delivering builds.

Protect systems from external and off-LAN network activity

To protect systems from potentially harmful network activity inbound and outbound network connections other than allowed URLs and necessary services should be blocked.

To assure that all source and other intellectual property on engineering machines is safeguarded, each system's cybersecurity defenses should be configured to prevent infiltration and exfiltration on all engineering workstations (e.g., configuring intrusion detection and prevention, behavior blocking, reputation-based security, machine learning-based protection, application isolation and control, and vulnerability protection).

Version control pipeline configurations

Pipeline configurations should be version controlled. Administrators should only update the configuration code, not the actual systems.

In a continuous delivery (CD) pipeline environment, the CD orchestrator should be the only entity that manages all the environments, for example the development and production environments. All

configurations and rules for the environments should be version-controlled and managed by the orchestrator. This will ensure any changes—whether malicious or accidental—that could weaken the security posture of the system will be immediately visible.

Administrators should rarely have to adjust the systems themselves, as configuration settings are in code and executed by the pipeline. An exception to this would be when an administrator has to fix the mean time to failure in production. In this case, the secrets management tool can generate a temporary Socket Shell (SSH) key for a limited time to allow the administrator access. Once approved, administrative changes should be adjusted in the pipeline configurations and automatically managed.

Also ensure that the administrator tools are not in the public environment such as a Kubernetes cluster. Utilize hardening guides such as the *Kubernetes Security - Operating Kubernetes Clusters and Applications Safely*[21] *and Kubernetes Hardening Guidance*[22]

Support the separation of duties. For example, the lead or business owner should be the owner and administrator of the build keys. The root account should not have access to the key.

Multi-factor authentication

Each system should use multi-factor authentication (MFA): wherever possible, require MFA for access to build pipeline systems. Limit access to build pipeline machines using best practices such as role-based access control and least privilege. For more information on this, please refer to *NIST SP 800-172 sec. 3.1, Role Based Access Control.* This specifically explains how to employ dual authorization to execute critical or sensitive system and organizational operations.

Segregate the engineering network

Each system should only be accessible via an engineering network that is completely segregated from the organization's corporate network. If possible, the engineering network should have no direct access to the Internet.

Minimize and regularly audit service accounts

The use of service accounts, like non-human privileged accounts used to run automated processes, should be minimized and carefully audited. Every service account login should be logged. These logs should include date and time and the origin of login. Service accounts should follow the "least privileged" policy. All service accounts should be regularly reviewed to assure they are still needed, and unnecessary accounts removed. Per guidelines in *NIST SP 800-53*, software function privileges should be raised when needed to perform a function and then lowered when completed.

Log all access to the build pipeline

All access to build pipeline systems should be logged. At minimum, log the MFA ID of the user authenticating access and the date and time.

---

[21] https://kubernetes-security.info/

[22] https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance/

Protect any secrets associated with the build pipeline

Best practices should be implemented to protect any secrets associated with the build pipeline. Secrets can be but are not limited to account passwords, API keys, and private certificates. Require default usernames and passwords in the pipeline to be changed. Any documentation inside the organization containing secrets should have strong role-based access control. Additionally, avoid putting secrets in plain text in any code (e.g., automation code), avoid logging sensitive data in application logs, and regularly rotate secrets. Please refer to *NIST SP 800-57, Part 1, Revision 5* for more details.

Recommended Mitigations (advanced)

The following mitigations describe advanced build best practices, and may offer additional protection when they complement the mitigations described previously in Section **2.4.1 Build Chain Exploits**.

Hermetic builds

All transitive build steps, sources, and dependencies should be fully declared up front with immutable references and the build steps should run with no network access.

The developer-defined build script must declare all dependencies, including sources and other build steps, using immutable references in a format that the build service understands.

The build service:

- Must fetch all artifacts in a trusted control plane,
- Must **not** allow mutable references,
- Must verify the integrity of each artifact,
- Must prevent network access while running the build steps.

If the immutable reference includes a cryptographic hash, the service must verify the hash and reject the fetch if the verification fails. Otherwise, the service must fetch the artifact over a channel that ensures transport integrity, such as Transport Layer Security (TLS) or code signing.

A "best effort" is sufficient when attempting to prevent network access while running build steps. This should deter a reasonable team from having a non-hermetic build, but it need not stop a determined adversary. For example, using a container to prevent network access is sufficient.

Reproducible builds

Reproducible builds provide additional protection and validation against attempts to compromise build systems. They ensure the binary products of each build system match: i.e., they are built from the same source, regardless of variable metadata such as the order of input files, timestamps, locales, and paths. Reproducible builds are those where re-running the build steps with identical input artifacts results in bit-for-bit identical output. Builds that cannot meet this must provide a justification why the build cannot be made reproducible.

- Establish and maintain an authoritative source and repository to provide a trusted source and accountability for approved and implemented system components as defined in NIST SP800-172 Sec. 3.4,

- Employ automated mechanisms to detect misconfigured or unauthorized system components; after detection, remediation is performed to either patch, re-configure or remove the identified components,
- Employ automated discovery and management tools to maintain an up-to-date, complete, accurate, and readily available inventory of system components,
- Identify and authenticate as defined in NIST SP800-172 *Sec. 3.5 [Assignment: organization-defined systems and system components*] before establishing a network connection using bidirectional authentication that is cryptographically based and replay resistant,
- Employ automated mechanisms for the generation, protection, rotation, and management of passwords for systems and system components that do not support multi-factor authentication or complex account management,
- Employ automated or manual/procedural mechanisms to prohibit system components from connecting to organizational systems unless the components are known, authenticated, in a properly configured state, or in a trust profile,
- Employ a means to allow the comparison of binaries built from two or more disparate, segmented, protected, and secured environments.

These mitigations can be modeled after the emerging frameworks, including build requirements of the Supply-Chain Levels for Software Artifacts (SLSA) project and the Software Component Verification Standard (SCVS). SLSA provides for different security levels, each of which provide requirements, processes, and best practices to increase trust in software. SCVS is set by the Open Web Application Security Project (OWASP). The standards comprise six families of control requirements including build environment for verification of the integrity of software supply chain.

The artifacts for builds should include, at a minimum, the source repository, the third-party dependencies, the build script, and the output of the build. Some products may supply these artifacts to the recipient of the software, though in many cases the supplier may require a special license or agreement to obtain these artifacts. For reproducible builds, the artifact should be the output of the script that compares the builds. All artifacts should be retained by the supplier for the entire support lifetime of the product until it is marked for end of life.

Recommended Mitigations

Advanced techniques may include:

1. SSDF PO.3 (Implement a Supporting Toolchain), specifically PO.3.2 and PO.3.3, since the toolchain is used in the build process.
2. SSDF PO.4 (Define Criteria for Software Security Checks), specifically PO.4.2 to gather information from the build to support security criteria.
3. SSDF PS.1 (Protect All Forms of Code from Unauthorized Access), specifically PS.1.1, generating the information in PS.2.1 and implementing PS.3.1.
4. SSDF PW.6 (Configure the Compilation and Build Processes).
5. SSDF PW.8 (Test Executable Code) if the tests are designed to be run and verified as part of the build process.

6. SSDF PW.9 (Configure the Software to Have Secure Settings by Default), particularly as part of the architecting, implementing and maintaining the build process.
7. SSDF RV.1 (Identify and Confirm Vulnerabilities on an Ongoing Basis) particularly RV.1.2 where automated code scanning may be part of the build process.

2.4.2 Exploited Signing Server

Software distributed as an artifact to a customer should be delivered with a unique cryptographic signature which verifies the integrity of the software artifact. However, if the signing facility itself has been compromised, then the delivered artifact may also have been compromised, and the signature validates the compromised artifact and not the true artifact.

***NOTE**: Software delivered to a customer as a service rather than as a binary artifact will not typically be delivered with a cryptographic signature.*

A threat actor could impersonate a target by compromising the code-signing service, using the signing system to sign compromised components or products. They could do this by leveraging misconfigured account access controls on a server, or exploiting the service using a known or zero-day exploit.

A threat actor could also impersonate a target using a self-signed certificate and injecting into the build or signing process.

Recommended mitigations

Code signing is usually the last line of defense against a software supply chain exploit. Both suppliers and developers work together to ensure the integrity of the signing servers is not compromised. Section **2.2.1 Protect All Forms of Code from Unauthorized Access** of Part 2 of this series, focusing on the supplier, will discuss the high-level supplier specific concerns. The following procedures are used by developers to ensure code-signing integrity:

1. Implement strong authentication methods such as strong passwords, certificates, two factor authentication (MFA), and physical access control to protect the signing infrastructure.
2. Control user access to the signing infrastructure, using least privileges, revocation of user credentials after departure or termination, MFA for code commits, and continuous authentication utilizing behavior analytics.
3. Conduct code signing on a physically isolated network segment.
4. Use intrusion detection and protection systems in the code-signing environment to protect the code-signing resources, machines, and process used.
5. Deploy and use a Security Information and Event Management (SIEM) system.
6. Use cryptography in transit and at rest.
7. Apply hardening procedures on the signing environment systems that allow customers to deploy and install only signed and verified release packages and products.
8. Use centralized log servers (with append only, encryption, etc.)
9. Ensure the signing system meets baseline security standards.
10. Require multi-party approval for physical and remote access and log all access.

11. Grant super-user access to only a small number of signing system admins.
12. Implement the following Indirect Controls:
    a) Perform periodic vulnerability scans (network and web application),
    b) Perform periodic penetration testing (network, web, wireless and red teaming),
    c) Classify documents properly (Confidential, Top Secret, etc.),
    d) Watermark usage on the documents,
    e) Use Data Loss Prevention (DLP) tools,
    f) Properly dispose of physical media by destroying it,
    g) Properly dispose of digital media by wiping it,
    h) Monitor / perform integrity checks of executables, libraries, configuration files, etc.

2.5 Deliver Code

Software suppliers utilize distribution systems to deliver software packages to customers. The package includes metadata (e.g., a version number) and software to install or update in the customer's information system and network. Before shipping the package to the customers, the developer should perform binary composition analysis to verify the contents of the package.

The distribution system is comprised of a repository, which stores packages for delivery, and a package manager at the customer information system. The two entities establish a secure connection over the internet and transfer the package.

2.5.1 Final Package Validation

The final package or update to be delivered to a customer may have issues that expose the developer and customers to cybersecurity and privacy risks. For example, it may contain confidential information (e.g., hard coded credentials, personal data), open source software license issues, and components included in files with unknown origin. Moreover, the deliverable may have been built with improper compiler options or build settings.

Recommended mitigations

1. Binary software composition analysis tools can investigate what exactly is included in the final deliverables and identify potential issues in the final packages described above. The developer should run a binary scanning or composition analysis tool and ensure the integrity of its product before delivery. The tool can detect potential vulnerabilities and threats – including software of unknown provenance (SOUP) and secrets inadvertently included in the final packages – and produce an SBOM of the final package for the customer.
2. The organization can compare the binary analysis output and the other artifacts from the build process to ensure that the final package includes only the intended software components. Upon receipt, the customer can run the binary software composition tool to assess risks by verifying the contents of the delivered code before the deployment. The customer can continue to utilize the tool for continuous monitoring of post deployment vulnerabilities.
3. Supplier and customer run binary scanning or composition analysis tools to verify the integrity of final software packages or updates provided by developers. The developer can

include the SBOMs within the final packages. It may depend on contracts or arrangements between developer and customers. An SBOM should contain all primary (top level) components of the final product, with all their transitive dependencies listed. Depending on the contracts with customers, the supplier, or the developer provides further details.

2.5.2 Potential Tactics to Compromise the Software Packages and Updates

The package may be compromised while going through the distribution system from the supplier to the customer. An adversary may attempt a man-in-the-middle attack or compromise the source code repository. As a result, malware or vulnerabilities can be introduced to the package, or an older version of the package containing an exploitable vulnerability can be delivered to the customer. Installing the compromised packages will impose risk to the customer organization and its system.

Recommended mitigations

The package including the correct metadata should be signed by a cryptographically-secure signature algorithm before being uploaded to the repository. Alternatively, the package can include cryptographic hashes. The package manager should verify the signature or hashes and the metadata including the version number. If verification fails, the manager should not process the package.

Note: *The developer must take great care to protect its private key used for the digital signature or hashes as anyone can impersonate the developer if the private key is stolen. A lost key may prevent future updates or incur challenges with distributing new keys.*

1. Product-Level:
    a) Hash/Digital Signature of Product Distribution Package,
    b) Hash/ Digital Signature of Product Update,
    c) Hash/ Digital Signature of Product Upgrade.
2. Component-Level:
    a) Hash/ Digital Signature of Product Components in Distribution Package.

Note: *Further study is necessary to define what types of components or static files should be signed or hashed in a package. The static files can include SBOM, configuration files and documentation. For correct verification of such components, guidelines or standards are needed to clearly specify how to create the signature and hash.*

2.5.3 Compromises of the Distribution System

Attacks to the distribution system include compromising the repository to introduce malware into the packages stored in the repository, taking advantage of the package manager vulnerabilities to direct it to a malicious site, and a MITM attack between the supplier, the repository, and the package manager. As a result, a compromised package can be delivered to the customer.

Recommended mitigations

The following activities may be optional if the developer takes the mitigation measures described in section **2.5.2 Potential Attacks to Compromise the Software Packages and Updates**. The

developer should secure the repository according to information security management standards or guidelines to ensure its integrity. For example, apply continuous monitoring to prevent, detect and remediate threats against the repository. The developer should practice the recommendations described in the proceeding sections of Section **2.5 Deliver Code** of this document to develop, test, and provide a secure package manager.

In addition, the developer, or the supplier should manage new vulnerabilities associated with the package manager and ensure that the customer uses its latest version. Moreover, the developer should ensure that transport layer security of the distribution system is configured properly to ensure the confidentiality, integrity, and authenticity of information to be transferred, for example, the TLS protocol version, algorithms for key exchange, encryption, message authentication and signature as recommended by NIST SP 800-52 rev.2 (*Guidelines for the Selection, configuration, and Use of Transport Layer Security (TLS) Implementations*).

1. Protect All Forms of Code from Unauthorized Access (SSDF PS.1). As described in the previous subsection, the developer may take the following mitigation measures:
    a) Repositories: apply information security management standards or guidelines to secure the repositories,
    b) Package managers: apply secure development process to provide secure and up-to-date package managers to the customer,
    c) Distribution system transport layer security: select, configure, and use transport security implementation recommended by NIST SP 800-52 rev.2.

3 Appendices

3.1 Appendix A: Crosswalk between Scenarios and SSDF

The reference numbers in the below crosswalk may look similar for each role (Developer, Supplier and Customer), however they correlate to their specific part of this series.

SSDF #	Developer	Supplier	Customer
PO.1	2.2.3 Secure Development Practices	2.1.1 Define criteria for software security checks	
PS.1	2.2.1.1 Source Control Check-in Process 2.2.1.4 Code Reviews 2.2.6 External Development Extensions 2.3.2 Selections and Integration 24.1 Build Chain Exploits 2.5.3 Secure the Distribution System	2.2.1 Protect all forms of code from unauthorized access 2.2.2 Provide a mechanism for verifying software release integrity (PS.1, PW.9)	
PS.3	2.2.1.1 Source Control Check-in Process 2.2.1.2 Automatic and Manual Dynamic and Static Security / Vulnerability Scanning 2.3.2 Selections and Integration 2.3.3 Obtain Components from a Known and Trusted Supplier 2.4.1 Build Chain Exploits	2.2.3 Archive and protect each software release	
PW.1	2.3.2 Selections and Integration	2.3.1 Design software to meet security requirements	
PW.3	2.2.3 Secure Development Practices 2.3.2 Selections and Integration 2.3.3 Obtain Components from a Known and Trusted Supplier	2.3.2 Verify third-party software complies with security requirements	2.1 Procurement/Acquisition (1) Requirements Definition / Recommended Controls (viii)(viii) 2.2 Deployment (6) (2) Testing – Functionality (c) Recommended Controls (ii) Verify contents in SBOM

	2.3.4 Component Maintenance 2.3.5 Software Bill of Material (SBOM)		2.2 Deployment (6) Deploy (3) Contracting / Recommended Controls (v) (viii) (ix)(x)
PW.6	2.2.3.2 Use of Unsecure Development Build Configurations 2.4.1 Build Chain Exploits	2.3.3 Configure the compilation and build processes	
PW.7	2.2.1.4 Code Reviews 2.2 Open source Management Practices 2.2.6 External Development Extensions 23.2 Selections and Integration 2.3.3 Obtain Components from a Known and Trusted Supplier	2.3.4 Review and/or analyze human-readable code	
PW.8	2.2.1.3 Nightly Builds with Regression Test Automation 2.3.2 Selections and Integration 2.4.1 Build Chain Exploits	2.3.5 Test executable code	
PW.9	2.2.3.2 Use of Unsecure Development Build Configurations 2.4.1 Build Chain Exploits	2.2.2 Provide a mechanism for verifying software release integrity (PS.1, PW.9) 2.3.6 Configure the software to have secure settings by default	
RV.1	2.3.4 Component Maintenance 2.4.1 Build Chain Exploits	2.4.1 Identify, analyze, and remediate vulnerabilities on a continuous basis	

3.2 Appendix B: Dependencies

Green - Dependencies/artifacts recommended to be provided by the supplier for benefit of the developer.

Dark Green - Dependencies/artifacts recommended to be provided by third-Party suppliers for benefit of the developer.

Pink - Dependencies/artifacts recommended to be provided by the customer for benefit of the supplier/developer.

#	Dependency
1	Provide issues from customers
2	Provide given hashes as required
3	SDLC policies and procedures
4	Secure architecture, high-level design
5	Qualified team assembly with code/security training
6	Independent QA individual/team
7	Independent security audit individual/team
8	Open source Review Board (OSRB) with repository
9	Product release management/resources
10	SBOM
11	Development location and information
12	Third-party SBOM
13	Third-party License
14	Release notes (detailing vulnerabilities fixed)
15	Vulnerability notifications
16	Publish updates and patches to the customer to address new vulnerabilities or weaknesses found within the product
17	Requirements and criteria for success
18	Implied industry security requirements
19	Provide issues from operational environment, take updates and patches
20	Vulnerability notifications and reporting from the users

3.3 Appendix C: Supply Chain Levels for Software Artifacts (SLSA)

Supply-Chain Levels for Software Artifacts (SLSA) is a security framework from source to service, giving anyone working with software a common language for increasing levels of software security. The framework is currently in Alpha stage and constantly being improved by supplier-neutral community. Google has been using an internal version of SLSA since 2013 and requires it for all of their production workloads. http://slsa.dev

Requirement	Description	L1	L2	L3	L4
Scripted build	All build steps were fully defined in some sort of "build script". The only manual command, if any, was to invoke the build script. Examples: • Build script is Makefile, invoked via make all. • Build script is. github / workflows / build.yaml, invoked by GitHub Actions.	✓	✓	✓	✓
Build service	All build steps ran using some build service, not on a developer's workstation. Examples: GitHub Actions, Google Cloud Build, Travis CI.		✓	✓	✓
Ephemeral environment	The build service ensured that the build steps ran in an ephemeral environment, such as a container or VM, provisioned solely for this build, and not reused from a prior build.			✓	✓
Isolated	The build service ensured that the build steps ran in an isolated environment free of influence from other build instances, whether prior or concurrent. • It MUST NOT be possible for a build to access any secrets of the build service, such as the provenance signing key. • It MUST NOT be possible for two builds that overlap in time to influence one another. • It MUST NOT be possible for one build to persist or influence the build environment of a subsequent build. • Build caches, if used, MUST be purely content-addressable to prevent tampering.			✓	✓
Parameterless	The build output cannot be affected by user parameters other than the build entry point and the top-level source location. In other words, the build is fully defined through the build script and nothing else. Examples: • GitHub Actions workflow dispatch inputs MUST be empty.				✓

	• Google Cloud Build user-defined substitutions MUST be empty. (Default substitutions, whose values are defined by the server, are acceptable.)				

Hermetic	All transitive build steps, sources, and dependencies were fully declared up front with immutable references, and the build steps ran with no network access.
	The developer-defined build script:
	• MUST declare all dependencies, including sources and other build steps, using immutable references in a format that the build service understands.
	The build service:
	• MUST fetch all artifacts in a trusted control plane.
	• MUST NOT allow mutable references.
	• MUST verify the integrity of each artifact.
	  o If the immutable reference includes a cryptographic hash, the service MUST verify the hash and reject the fetch if the verification fails.
	  o Otherwise, the service MUST fetch the artifact over a channel that ensures transport integrity, such as TLS or code signing.
	• MUST prevent network access while running the build steps.
	  o This requirement is "best effort." It SHOULD deter a reasonable team from having a non-hermetic build, but it need not stop a determined adversary. For example, using a container to prevent network access is sufficient.
					✓

Reproducible	Re-running the build steps with identical input artifacts results in bit-for-bit identical output. Builds that cannot meet this MUST provide a justification why the build cannot be made reproducible. "○" means that this requirement is "best effort". The developer-provided build script SHOULD declare whether the build is intended to be reproducible or a justification why not. The build service MAY blindly propagate this intent without verifying reproducibility. A customer MAY reject the build if it does not reproduce.				○

3.4 Appendix D: Artifacts and Checklist

In principle, any artifacts created during the lifecycle of the software development process are owned by a developing organization. These organizations can determine what artifacts are made available with potential and current customers of a product with or without a Non-Disclosure Agreement (NDA). Availability of information must take into consideration regulatory and legal requirements, the customer requirements for the information and the risk involved by exposing information leading to the exploitation of the product. Exceptions may include open source development organizations, which are more inclined to make all development information available, to include source code.

When defining the availability of an artifact, the general terms used in this section will be the following:

1. Publicly disclosed,
2. Externally available:
    a) under a Non-Disclosure Agreement (NDA),
    b) government agency mandated requirement.
3. Private / company confidential.

The availability of an artifact varies between companies and agencies and is only described here as a reference for what might be possible when using artifacts to validate the software supply chain process. Some artifacts, such as a high-level architecture document may be intentionally generated to allow any perspective consumers an introductory artifact detailing the overall strategies used in the design, development, and operation of a product. These publicly disclosed documents may describe common industry nomenclature, such as Federal Information Process Standards (FIPS) compliance, cryptography standards used, development processes adhered to or certifications processes passed. NDA and government mandated availability require contractual agreements providing access to artifacts that would not normally be exposed by the organization that produced the product. While private or company confidential artifacts are generally low-level and detailed work products that may contain sensitive secrets and knowhow and if exposed, provide potential insight into product's competitive implementation and threat vectors that may not be addressed in the product, therefor posing a threat if exposed outside of the producers environment. Private/company confidential artifacts are generally maintained by the "Suppliers" and "Developers" of the product to facilitate the auditing and validation of adherence to the Secure Software Development Lifecycle (Secure SDLC) and security practices set forth by the product owner, company, or organization. For more information on the Secure SDLC process, refer to Section **2.1 "Secure Product Criteria and Management,"** subsection Recommended Mitigations, Item 8.

Most of the artifacts collected during the development lifecycle are not meant to be shared outside the developing organization yet may be preserved in persistent storage as evidence to verify the integrity of the policies and processes used during the development of a product. A developer should securely retain artifacts of software development for a certain duration according to the secure software development policies and processes. As a by-product of the process used to implement and mitigate the attack surface and threat model of the software as well as the software build pipeline during the development process, the following artifacts may be created, and collected:

Artifact Examples	Description/Purpose
High-level Secure Development Lifecycle Process document	Attestation to secure development practices which can cover: • Secure software architecture/design process • Attack surface investigation and threat modeling process • Secure software development/programming training • Software security testing process • Source control check-in process • Trusted repository for modules and processes • Continuous integration and delivery (CI/CD) processes • Defect/vulnerability reporting and customer update process • Code review process for security and continuous software security improvement • Continuous verification of third-party binaries • Open source management practices • Hardening the build environment • Secure relationship with a third-party supplier • Process to secure the signing server • Final package validation process
Product Readiness checklist	Attestation to product release, product readiness for shipment, and secure shipping criteria which can cover: • No pending known critical bugs and vulnerabilities (e.g. bug track report) • Cryptographically signed components • Proper software licensing
Product Support/Response Plan	Attestation to vulnerability discloser and response process (e.g. handling of policy violation and anomalies)
Software Bill of Material (SBOM)	• Attestation to the integrity of the producer • Attestation to the security and authenticity of components included in the product • Attestation to the third-party software components • Attestation to the integrity of software licenses
Architecture/Design Documents	• Attestation to secure architecture/design practices • Mitigation of attack surface vulnerabilities • Attestation to mapping secure requirements to software architecture and components
Developer Training Certificates/Training Completion Statistics/data	• Attestation to secure development practices • Attestation to secure coding practices
Threat Model Results Document	• Attestation to secure design practices • Attestation to secure third-party component integration practices

High-level Software Security Test Plan and Results	High-level, system and unit level test plan and results (A set of tests should be commensurate with the requirements and risk profile of the product or service.) • Coverage details • SAST - Static Application Security Testing • DAST - Dynamic Application Security Testing • SCA - Software Composition Analysis • Fuzzing/Dynamic • Penetration • Red team testing • Black box testing • QA security feature analysis
Automatic and Manual Dynamic and Static Security/Vulnerability Reports (Security Scanning Results) Reports	The reports can cover: • Security Scanning Results for Static, Dynamic, Software Composition Analysis and Fuzzing • Security Scanning Results for Penetration or Red-Teaming • Attestation to secure development/build/test practices • Mitigation against known software weakness classes in the Common Weakness Enumeration (CWE) • Mitigation against publicly known vulnerabilities and Common Vulnerabilities and Exposures (CVEs)
Open source Review Process Document and Allowed List	Attestation to secure open source review process and management
Build Log	• Attestation to the integrity of securely built products • Attestation to no known critical errors/warnings • Attestation to use of tool-chain defenses (stack checking, ASLR, etc.)
Secure Development Build Configurations Listing	• Attestation to secure build environment
Third-Party Software Tool-Chains List	• Attestation to secure build environment

The artifacts described in the table above may be used for attestation of the integrity of an organization's secure development process that was used to produce a given product. Organizations can then provide a high-level checklist, illustrated below, which may utilize artifacts created during the development process that attest to the adherence, at some level, to the recommended practice during the development process. The developer may add a brief description regarding how the organization supports a check list item in addition to Yes/No/Not Applicable (NA)/Incomplete (Inc) response, e.g. alternative practices to support it and reasons for non-applicability.

The document references in the below table are focused on the Developer portion of this series.

Measurable Outcome/ Description	Practice Observed Yes, No, NA, Inc	SSDF Tasks	Artifact Examples	Document References
Secure Product Criteria & Management				
Do you define policies that specify risk-based software architecture and design requirements?		**PO.1.2**	Architecture/Design Documents	2.1 Secure Product Criteria and Management
Do you require team members to regularly participate in secure software architecture, design, development, and testing training and monitor their training completion?		**PO.2.2** **RV.3.4**	Training Completion Data/Statistics Developer Training Certificates	2.1 Secure Product Criteria and Management 2.2.1 Secure Software Development/ Programming Training
Have development team members attended training programs specific to their roles, development tools and languages to update their skills?		**PO.2.2**	Training Completion Data/Statistics Developer Training Certificates	2.2.1 Malicious Modification of Source Code Threat Scenario, Subsection 6: Secure Product Criteria and Management
At a minimum, for all critical software components and external services that your team operates and owns, have you completed the attack surface survey and threat models for all such services?		**PW.1.1** **PW.2.1**	Threat Model Results Documents	2.1 Secure Product Criteria and Management 2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 5. Requirements to Design / Development Feature Mapping
Do you have up to date threat models for all critical components your team ships that have been reviewed by a person trained in software security and do you make this document available to other teams that pick up your component?		**PW.1.1, PW.2.1**	Threat Model Results Document	2.1 Secure Product Criteria and Management

Has your team held a black-box investigation for security?			Black box test results	2.1 Secure Product Criteria and Management 2.3.1 Third-Party Binaries Threat Scenario
Do you have and use security tools and methodology (e.g. recommended by NISTIR 8397) for static, dynamic and Software Composition Analysis and ensure that all high severity issues are addressed?		**PO.3.1**	SAST, DAST, SCA test results	2.1 Secure Product Criteria and Management 2.3.2 Selections and Integration Threat Scenario
Do you perform input fuzzing as part of a regular process for your component or product's inputs?		**PW.8.2**	Fuzzing/Dynamic test results	2.1 Secure Product Criteria and Management
Do you have security testing as part of your overall QA plan to enhance the testing of specific features of your product?			Product test results	2.1 Secure Product Criteria and Management
Has your product or components been identified as needing penetration testing? If so, are all issues found recorded in a bug tracker, with high priority defects set to prevent shipment of the product?		**PW.8.2**	Penetration Test Results	2.1 Secure Product Criteria and Management
Has your product or components been identified as needing red-team testing? If so, are all issues found recorded in a bug tracker, with high priority defects set to prevent shipment of the product?			Red-Team Test Results	2.4.3 Signing Server Exploits Threat Scenario
Has your product or components been identified as needing testing for security gaps by an external party? If so, has your code or systems been tested for security gaps by an external party (e.g. JFAC Software Assurance providers, pen testing, threat model			Third-party Test Results	2.1 Secure Product Criteria and Management

reviews, vulnerability scan tools and red-teams)?				
Does your release include an SBOM and confirmation that no unacceptable security vulnerabilities are pending, binaries are digitally signed and meet cryptographic standards?			SBOM Product Bug Tracking Report	2.1 Secure Product Criteria and Management 2.2.5 Defect/Vulnerability Customer Issues Report Scenario 2.2.6 External Developer Extensions Threat Scenario 2.3.5 Software Bill of Material (SBOM) Threat Scenario 2.5.1 Final Package Validation Threat Scenario
Are all public cloud resources continuously monitored by a tool that analyzes and alerts for policy violations and anomalies?			Product Support / Response Plan	2.1 Secure Product Criteria and Management 2.2.6 External Development Extensions Threat Scenario
Are the alerts being actively monitored?			Product Support / Response Plan	2.2.5 Defect/ Vulnerability Customer Issues Report Scenario
Is there a process in place to resolve policy violations within a specific amount of time?			Product Support / Response Plan	2.1 Secure Product Criteria and Management 2.2 Develop Secure Code 2.2.5 Defect/Vulnerability Customer Issues Report Scenario
Develop Secure Code				

Are all of your security issues tracked with a bug tracker and scored, for example using CVSSv3 scores to help determine fix prioritization and release scheduling?		**RV.2.1**	Secure Software Development Lifecycle Process document Bug Tracker Report	2.1 Secure Product Criteria and Management 2.2.1 Malicious Modification of Source Code Threat Scenario 2.2.1.2. Automatic and Manual Dynamic and Static Security/Vulnerability Scanning
Do you use access-controlled applications to store sensitive vulnerability information for all issues affecting production code that is more restrictive than plain bug tracker defects?		**PO.5.1**	Secure Software Development Lifecycle Process document	2.4.1 Build Chain Exploits Threat Scenario
Does your team have a process to reduce a class of vulnerabilities based on previously identified vulnerabilities or incidents?		**PW.7.2**	Secure Software Development Lifecycle Process document	2.2.1 Malicious Modification of Source Code Threat Scenario 2.2.5 Defect/Vulnerability Customer Issue Reports Threat Scenario
Do you perform nightly builds with automated regression and security test to quickly detect problems with recent builds?			Secure Software Development Lifecycle Process document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 3: Nightly Builds with Regression Test Automation Plan
Are code check-ins gated by code collaborators and source control to prevent anyone from accidentally or intentionally submitting un-reviewed code changes?		**PW.7.2**	Secure Software Development Lifecycle Process document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 1: Source Control Check-In Process 2.3.1 Malicious Modification of Source Code Threat Scenario 2.2.1 Malicious Modification of Source Code Threat Scenario: Subsection 6. Secure Software Development/ Programming Training

Does the team require code reviews for all code and build scripts / configuration changes?		**PW.7**	Secure Software Development Lifecycle Process document	2.4.1 Build Chain Exploits Threat Scenario
Does the team measure and analyze the quality of the code review process?			Secure Software Development Lifecycle Process document	2.1 Secure Product Criteria and Management 2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 4. Code Reviews
Do you ensure only required modules are included in the product and "unused" modules and code out of scope of the requirements and design document are uninstalled or removed, mitigating "living-off-the-land" attacks and decreasing the attack surface?			Secure Software Development Lifecycle Process document Requirements Document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 5: Requirements to Design/ Development Feature Mapping
Do you map all your security requirements to the software component of the product and track their completion/adherence?			Secure Software Development Lifecycle Process document Security Requirements Document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 5: Requirements to Design/ Development Feature Mapping
Are unmodified third-party libraries retrieved from a common location such as a secured persistent storage or shared repository location out of band of the development process and not individually built by your team?			Secure Software Development Lifecycle Process document	2.1 Secure Product Criteria and Management 2.2.4 Code Integration Threat Scenario 2.3.1 Third-Party Binaries Threat Scenario 2.3.3 Obtain Components from a Known and Trusted Supplier Threat Scenario

				2.3.4 Component Maintenance Threat Scenario
Do you monitor new vulnerabilities applicable to your software e.g. using registered vulnerability notification services?		**RV.1.1**	Secure Software Development Lifecycle Process document	2.3 Verify Third-Party Components 2.5.1 Final Package Validation Threat Scenario
Do you have and adhere to responsible disclosure requirements for all externally identified vulnerabilities?			Secure Software Development Lifecycle Process document	2.2.5 Detect / Vulnerability Customer Issue Reports Threat Scenario
Are all of your builds continuously built and tested?			Secure Software Development Lifecycle Process document	2.4.2 Build Chain Exploits; Advanced Practices Threat Scenario
Does a check-in immediately trigger a build?			Secure Software Development Lifecycle Process document	2.4.2 Build Chain Exploits; Advanced Practices Threat Scenario
Does a completed build automatically go through some acceptance testing?			Secure Software Development Lifecycle Process document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 3: Nightly Builds with Regression Test Automation
If the testing passes, is the build automatically deployed so others can consume it?		**PO.3.1**	Secure Software Development Lifecycle Process document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 3: Nightly Builds with Regression Test Automation
Verify Third-Party Components				
Do you track all third-party components you use directly and all internal components in a secure and persistent repository?		**PS.1.1** **PW.4.1**	Secure Software Development Lifecycle Process document OSRB Approved List Product/Component Scan Results	2.2.2 Open source Management Practices 2.3.1 Third-Party Binaries Threat Scenario
Do you have the requirement for an Open source Review Board to approve third-party		**PW.4.1** **PW.4.4**	Secure Software Development	2.3.3 Obtain Components from a Known and Trusted

libraries included in a product and audit approved third-party libraries for version adherence and vulnerabilities?			Lifecycle Process document OSRB Approved List	Supplier Threat Scenario
Do you remove or mitigate critical known vulnerabilities or end of life issues of third-party components before each release?		**PW.4.5**	Secure Software Development Lifecycle Process document OSRB Approved List	2.2.4 Code Integration Threat Scenario
When considering the selection of a third-party component, do use a known and trusted supplier that has a proven record for secure coding practices and quality delivery of their components?		**PO.1.3**	Secure Software Development Lifecycle Process document OSRB Approved List	2.3.3 Obtain Components from a Known and Trusted Supplier Threat Scenario
Within a developer environment, do you monitor and approve of all IDEs and third-party development/debugging extensions to ensure their adoption does not weaken the security posture of the local development environment?			Secure Software Development Lifecycle Process document	2.2.3 Use of Secure Third-Party Software Tool-Chains and Compatibility Libraries 2.2.6 External Development Extensions Threat Scenario
Do you have a trusted repository to support ongoing software composition analysis and security testing for all external and downloaded modules?			Secure Software Development Lifecycle Process document	2.2.4 Code Integration Threat Scenario
Harden the Build Environment				
Have you completed attack surface investigation and threat modeling for your build environment?			Threat/Risks Model Results Documents	2.1 Secure Product Criteria and Management 2.2.1 Malicious Modification of Source Code Threat Scenario: Subsection 7. Harden the Development Environment

Do you ensure that only in very rare cases, the build process accesses the open Internet and these cases are documented and approved within the security plan?		**PO.5.1**	Secure Software Development Lifecycle Process document	2.4.1 Build Chain Threat Scenario
Do you limit and secure access to your development environment to essential administrators?			Secure Software Development Lifecycle Process document	2.4 Harden the Development Environment
Do you monitor the build chain for unauthorized access and modifications?			Secure Software Development Lifecycle Process document	2.4 Harden the Development Environment
Do you document approval and audit logs of build chain modifications?			Secure Software Development Lifecycle Process document	2.4 Harden the Development Environment
Do you enforce build-chain configuration defensive techniques required to narrow the attack vectors of the components and products being developed?			Secure Software Development Lifecycle Process document Build Logs	2.2.3 Use of Secure Development Build Configurations
Do you ensure the integrity of the individual development environment, caring to harden the development systems within the build pipeline?			Secure Software Development Lifecycle Process document	2.2.3 Secure Developer Environment
Does your build process encrypt data in transit?			Secure Software Development Lifecycle Process document	2.5.3 Secure the Distribution System Threat Scenario
Does each critical server within the build chain owned by the team have a clearly defined owner responsible for patch maintenance?		**PO.5.1**	Secure Software Development Lifecycle Process document	2.1 Secure Product Criteria and Management 2.4.2 Build Chain Exploits; Advanced Practices Threat Scenario
Do you have a requirement that server patch levels are checked periodically?			Secure Software Development Lifecycle Process document	2.1 Secure Product Criteria and Management

Is unnecessary outbound internet connectivity blocked?		**PO.5.1**	Secure Software Development Lifecycle Process document	2.2.3 Secure Development Practices Threat Scenario Subsection 1: Secure the Developer Environment 2.4.1 Build Chain Exploits Threat Scenario
Is unnecessary inbound internet connectivity blocked?		**PO.5.1**	Secure Software Development Lifecycle Process document	2.2.3 Secure Development Practices Threat Scenario Subsection 1: Secure the Developer Environment 2.4.1 Build Chain Exploits Threat Scenario
Is the integrity of the builds verified to ensure no malicious changes have occurred during the build and packaging process, for example, are two or more builds performed in different protected environments and the results compared to ensure the integrity of the build process?			Secure Software Development Lifecycle Process document	2.4.2 Build Chain Exploits; Advanced Practices Threat Scenario
Do you use the toolchain to automatically gather information that informs security decision-making?		**P0.4.2**	Secure Software Development Lifecycle Process document	
Does the tool chain automatically scan for vulnerabilities and stop the build process and report errors when detected, if so configured?		**PS.1.1** **PW.7.2**	Secure Software Development Lifecycle Process document	2.2.1 Malicious Modification of Source Code Threat Scenario Subsection 2: Automatic and Manual Dynamic and Static Security / Vulnerability Scanning 2.2.4 Code Integration Threat Scenario
Do you store access credentials (e.g. hashes for passwords) and secrets in a secure (e.g. encrypted) location such as a secure vault?				2.4.1 Build Chain Exploits Threat Scenario
Secure Code Delivery				

Do you perform binary composition analysis of the final package?			Secure Software Development Lifecycle Process document	2.5.1 Final Package Validation Threat Scenario
Do you have a Software Bill of Materials (SBOM) that satisfies the contracts?		**PS.3.2** **PW.4.1**		2.5.1 Final Package Validation Threat Scenario
Do you digitally sign all required binaries you ship?		**PS.1.1** **PS.2.1**	Secure Software Development Lifecycle Process document	2.5.2 Instrument Integrity Checks, Code Signing and Hashing Threat Scenario
Do you ensure that no globally-trusted certificates are directly accessible and use a dedicated, protected signing server when signing is required?			Secure Software Development Lifecycle Process document	2.4.3 Signing Server Exploits
Are you using organization approved Configuration Management tools to sign your shipping binaries?			Secure Software Development Lifecycle Process document	2.4.2 Build Chain Exploits; Advanced Practices Threat Scenario
Do you comply with the use of cryptography recommended by the organization's security policy?		**PS.1.1**	Secure Software Development Lifecycle Process document	2.5.3 Secure the Distribution System Threat Scenario 2.4.3 Signing Server Exploits

3.5 Appendix E: Informative References

Abbreviation	Document Name
ACM	Communications of the ACM 17, "The Protection of Information in Computer Systems". Available at (http://web.mit.edu/Saltzer/www/publications/protection/index.html)
BSIMM10	Migues S, Steven J, Ware M (2019) Building Security in Maturity Model (BSIMM) Version 10. Available at (https://www.bsimm.com/download/)
CISA	Cybersecurity & Infrastructure Security Agency. Available at (https://www.cisa.gov/defining-insider-threats)
CISCO_SDLC	Cisco. 2021. Cisco Secure Development Lifecycle. Available at (https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/cisco-secure-development-lifecycle.pdf)
DoD CIO	DoD Enterprise, 2021. DevSecOps Fundamentals Lifecycle Phases Version 2.0. Available at (https://dodcio.defense.gov/Portals/0/Documents/Library/DoDEnterpriseDevSecOpsFundamentals.pdf)
EO14028	EOP. 2021. "Improving the Nation's Cybersecurity", Executive Order 14028, 86 FR 26633, Document number 2021- 10460. Available at (https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/)
FIPS140	National Institute of Standards and Technology. 2019. "Security Requirements for Cryptographic Modules." Available at (https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-3.pdf).
IDASOAR	Hong Fong EK, Wheeler D, Henninger A (2016) State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016. (Institute for Defense Analyses [IDA], Alexandria, VA), IDA Paper P-8005. Available at (https://www.ida.org/research-and-publications/publications/all/s/st/stateoftheartresources-soar-for-software-vulnerability-detection-test-and-evaluation-2016)
INTEL	Intel. Software Supply Chain Threats; A White Paper Version 1.0, July 2021. Available at https://www.intel.com/content/www/us/en/security/supply-chain-threat-whitepaper.html
ISO27034	International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), Information technology – Security techniques – Application security – Part 1: Overview and concepts, ISO/IEC 27034-1:2011, 2011. Available at (https://www.iso.org/standard/44378.html)
MITRE_CAPEC	MITRE. 2021. Common Attack Pattern Enumeration and Classification. Available at (https://capec.mitre.org/data/definitions/437.html)
MITRE_CVE	MITRE. 2021. "Common Vulnerability and Exposure, CVE." 2021. Available at (https://cve.mitre.org/index.html).

MSSDL	Microsoft (2019) Security Development Lifecycle. Available at https://www.microsoft.com/en-us/sdl
NASASTD8739	National Aeronautics and Space Administration. 2021. "SOFTWARE ASSURANCE AND SOFTWARE SAFETY STANDARD, NASA-STD-8739.8A." Available at (https://standards.nasa.gov/sites/default/files/standards/NASA/PUBLISHED/A1/nasa-std-8739.8a.pdf).
NICCS	National Initiative for Cybersecurity Careers and Studies, National Initiative for Cybersecurity Education. 2021 Workforce Framework for Cybersecurity (NICE Framework). Available at (https://niccs.cisa.gov/workforce-development/cyber-security-workforce-framework)
NISTCSF	National Institute of Standards and Technology. 2018. "Framework for Improving Critical Infrastructure Cybersecurity, Version 1.1." Available at (https://doi.org/10.6028/NIST.CSWP.04162018)
NISTMSDV	National Institute of Standards and Technology. 2018. "Guidelines on Minimum Standards for Developer Verification of Software". available at (https://www.nist.gov/system/files/documents/2021/07/13/Developer%20Verification%20of%20Software.pdf)
NTIASBOM	National Telecommunications and Information Administration. 2021. "The Minimum Elements for a Software Bill of Materials (SBOM)." Available at (https://www.ntia.doc.gov/report/2021/minimum-elements-software-bill-materials-sbom)
NVD	National Vulnerability Database. Available at (https://www.nist.gov/programs-projects/national-vulnerability-database-nvd)
OWASP_ASVS	Open Web Application Security Project (2019) OWASP Application Security Verification Standard 4.0. Available at https://github.com/OWASP/ASVS
OWASP_SAMM	Open Web Application Security Project (2017) Software Assurance Maturity Model Version 1.5. Available at (https://owasp.org/www-pdf-archive/SAMM_Core_V1-5_FINAL.pdf)
OWASP_SCVS	OWASP. 2021. "OWASP Software Component Verification Standard." Retrieved Sep. 25, 2021 (https://owasp.org/www-project-software-component-verification-standard/).
OWASP_TEST	Open Web Application Security Project (2014) OWASP Testing Guide 4.0. Available at (https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Testing_Guide_v4.pdf)
PCI_SSLRAP	Payment Card Industry (PCI) Security Standards Council (2019) Secure Software Lifecycle (Secure SLC) Requirements and Assessment Procedures Version 1.0. Available at (https://www.pcisecuritystandards.org/document_library?category=sware_sec#results)
SC_AGILE	Software Assurance Forum for Excellence in Code (2012) Practical Security Stories and Security Tasks for Agile Development Environments. Available at (http://www.safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf)

SC_FPSSD	Software Assurance Forum for Excellence in Code (2018) Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Lifecycle Program, Third Edition. Available at (https://safecode.org/wpcontent/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf)
SC_SIC	Software Assurance Forum for Excellence in Code (2010) Software Integrity Controls: An Assurance-Based Approach to Minimizing Risks in the Software Supply Chain. Available at (http://www.safecode.org/publication/SAFECode_Software_Integrity_Controls0610.pdf)
SC_TPC	Software Assurance Forum for Excellence in Code (2017) Managing Security Risks Inherent in the Use of Third-Party Components. Available at (https://www.safecode.org/wpcontent/uploads/2017/05/SAFECode_TPC_Whitepaper.pdf)
SC_TTM	Software Assurance Forum for Excellence in Code (2017) Tactical Threat Modeling. Available at (https://www.safecode.org/wpcontent/uploads/2017/05/SAFECode_TM_Whitepaper.pdf)
SLSA	The Linux Foundation. 2021. "Improving artifact integrity across the supply chain – SLSA." Available at (https://slsa.dev/)
SP80050	National Institute of Standards and Technology. 2021. "PRE-DRAFT Call for Comments: Building a Cybersecurity and Privacy Awareness and Training Program, SPS 800-50 Rev 1." Available at (https://csrc.nist.gov/publications/detail/sp/800-50/rev-1/draft)." Retrieved Sep. 25, 2021.
SP80052	National Institute of Standards and Technology. 2020. "Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations, SP 800-52 Rev. 2." Available at (https://csrc.nist.gov/publications/detail/sp/800-52/rev-2/final).
SP80053	National Institute of Standards and Technology. 2020. "Security and Privacy Controls for Information Systems and Organizations. Available at (https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final)
SP80057	National Institute of Standards and Technology. 2020. "Recommendation for Key Management: Part 1 – General, SP 800-57 Part 1 Rev. 5." Available at (https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final)
SP800160	National Institute of Standards and Technology. 2018. "Systems Security Engineering." Available at (https://doi.org/10.6028/NIST.SP.800-160v1)
SP800161	"National Institute of Standards and Technology. 2021. ""Supply Chain Risk Management Practices for Federal Information Systems and Organizations."" Available at (https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-161.pdf)"

SP800172	”Enhanced Security Requirements for Protecting Controlled Unclassified Information: A Supplement to NIST SP 800-171. Available at (https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-172.pdf)
SP800175B	National Institute of Standards and Technology. 2020. “Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms. SP 800-175B Rev. 1.” Available at (https://csrc.nist.gov/publications/detail/sp/800-175b/rev-1/final).
SP800181	National Institute of Standards and Technology, National Initiative for Cybersecurity Education. 2020. “Workforce Framework for Cybersecurity (NICE Framework).” Available at (https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-181r1.pdf)
SP800193	National Institute of Standards and Technology. 2018. “Platform Firmware Resiliency Guidelines, SP-800-193.” Available at (https://csrc.nist.gov/publications/detail/sp/800-193/final).
SP800207	National Institute of Standards and Technology. 2020. “Zero-Trust Architecture, SP-800-207.” https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf
SSDF	National Institute of Standards and Technology. 2020. “Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework (SSDF).” Available at (https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04232020.pdf)
SWEBOK3	IEEE Computer Society. 2014. Guide to the Software Engineering Body of Knowledge. Available at (https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3)
SYNOPSYS	Synopsys. 2021. “Synopsys Information Security Requirements for Vendors.” Available at https://www.synopsys.com/company/legal/info-security.html
ZDNET	IBM, ZDNET. 2021. “Managing a Software as a Vendor Relationship: Best Practices”. Available at (https://www.zdnet.com/article/managing-a-software-as-a-service-vendor-relationship-best-practices/)

3.6 Appendix F: Acronyms Used in This Document

Acronym	Meaning
API	Application Programming Interface
ASLR	Address Space Layout Randomization
CI/CD	Continuous Integration/Continuous Delivery
CNSSI	Committee on National Security Systems Instruction
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
DLP	Data Loss Prevention
DUNS	Data Universal Numbering System
EO	Executive Order
EOL	End of Life
ESF	Enduring Security Framework
FARS/DFARS	Federal Acquisition Regulation/Defense Federal Acquisition Regulation
FedRAMP	Federal Risk and Authorization Management Program
FIPS	Federal Information Process Standards
HIPAA	Health Insurance Portability and Accountability Act
HSM	Hardware Security Module
HTTPS	Hypertext Transfer Protocol Secure
IAST	Interactive Application Security Testing
IDE	Integrated Development Environment
LAN	Local Area Network
MFA	Multi Factor Authentication
MITM	Man in The Middle
ML	Machine Language
NIAP	National Information Assurance Partnership
NIST	National Institute of Standards and Technology (US DOC)
NTIA	National Telecommunications and Information Administration (US DOC)
NVD	National Vulnerability Database
OpenSSF	Open Source Security Foundation
OSRB	Open source Review Board
OSS	Open Source Software
OWASP	Open Web Application Security Project
PO	Prepare Organization

PS	Protect Software
PSIRT	Product Security Incident Response Team
PW	Produce Well-Secured Software
QA	Quality Assurance
RACI	Responsible, Accountable, Consulted, and Informed
RASP	Runtime Application Self-Protection
RM	Risk Management
ROP	Return-oriented Programming
RV	Respond to Vulnerabilities
SaaS	Software-as-a-Service
SAST	Static Application Security Testing
SBOM	Software Bill of Material
SCA	Software Composition Analysis
SCM	Supply Chain Management
SCM	Source Code Management
SCRM	Supply Chain Risk Management
SCVS	Software Component Verification Standard
SDLC	Software Development Lifecycle
SEH	Software Exception Handler
SHA	Secure Hash Algorithm
SIEM	Security Information and Event Management
SLSA	Supply-chain Levels for Software Artifacts
SOUP	Software of Unknown Provenance
SOX	Sarbanes-Oxley Act
SPDX	Software Package Data eXchange
SSDF	Secure Software Development Framework
SSH	Socket Shell
SwA	Software Assurance
SWID	Software Identification
TLS	Transport Layer Security
TSA	Time Stamp
UML	Unified Modeling Language
VAR	Value-added Reseller
VCS	Version Control System
VM	Virtual Machine
VPN	Virtual Private Network