Practices	Tasks	Notional Implementation Examples
Define Security Requirements for Software Development (PO.1): Ensure that security requirements for software development are known at all times so that they can be taken into account throughout the SDLC and duplication of effort can be minimized because the requirements information can be collected once and shared. This includes requirements from internal sources (e.g., the organization's policies, business objectives, and risk management strategy) and external sources (e.g., applicable laws and regulations).	**PO.1.1**: Identify and document all security requirements for the organization's software development infrastructures and processes, and maintain the requirements over time.	Example 1: Define policies for securing software development infrastructures and their components, including development endpoints, throughout the SDLC and maintaining that security. Example 2: Define policies for securing software development processes throughout the SDLC and maintaining that security, including for open-source and other third-party software components utilized by software being developed. Example 3: Review and update security requirements at least annually, or sooner if there are new requirements from internal or external sources, or a major security incident targeting software development infrastructure has occurred. Example 4: Educate affected individuals on impending changes to requirements.
	PO.1.2: Identify and document all security requirements for organization-developed software to meet, and maintain the requirements over time.	Example 1: Define policies that specify risk-based software architecture and design requirements, such as making code modular to facilitate code reuse and updates; isolating security components from other components during execution; avoiding undocumented commands and settings; and providing features that will aid software acquirers with the secure deployment, operation, and maintenance of the software. Example 2: Define policies that specify the security requirements for the organization's software, and verify compliance at key points in the SDLC (e.g., classes of software flaws verified by gates, responses to vulnerabilities discovered in released software). Example 3: Analyze the risk of applicable technology stacks (e.g., languages, environments, deployment models), and recommend or require the use of stacks that will reduce risk compared to others. Example 4: Define policies that specify what needs to be archived for each software release (e.g., code, package files, third-party libraries, documentation, data inventory) and how long it needs to be retained based on the SDLC model, software end-of-life, and other factors. Example 5: Ensure that policies cover the entire software life cycle, including notifying users of the impending end of software support and the date of software end-of-life. Example 6: Review all security requirements at least annually, or sooner if there are new requirements from internal or external sources, a major vulnerability is discovered in released software, or a major security incident targeting organization-developed software has occurred. Example 7: Establish and follow processes for handling requirement exception requests, including periodic reviews of all approved exceptions.

Practices	Tasks	Notional Implementation Examples
	PO.1.3: Communicate requirements to all third parties who will provide commercial software components to the organization for reuse by the organization's own software. [Formerly PW.3.1]	Example 1: Define a core set of security requirements for software components, and include it in acquisition documents, software contracts, and other agreements with third parties. Example 2: Define security-related criteria for selecting software; the criteria can include the third party's vulnerability disclosure program and product security incident response capabilities or the third party's adherence to organization-defined practices. Example 3: Require third parties to attest that their software complies with the organization's security requirements. Example 4: Require third parties to provide provenance data and integrity verification mechanisms for all components of their software. Example 5: Establish and follow processes to address risk when there are security requirements that third-party software components to be acquired do not meet; this should include periodic reviews of all approved exceptions to requirements.
Implement Roles and Responsibilities (PO.2): Ensure that everyone inside and outside of the organization involved in the SDLC is prepared to perform their SDLC-related roles and responsibilities throughout the SDLC.	**PO.2.1**: Create new roles and alter responsibilities for existing roles as needed to encompass all parts of the SDLC. Periodically review and maintain the defined roles and responsibilities, updating them as needed.	Example 1: Define SDLC-related roles and responsibilities for all members of the software development team. Example 2: Integrate the security roles into the software development team. Example 3: Define roles and responsibilities for cybersecurity staff, security champions, project managers and leads, senior management, software developers, software testers, software assurance leads and staff, product owners, operations and platform engineers, and others involved in the SDLC. Example 4: Conduct an annual review of all roles and responsibilities. Example 5: Educate affected individuals on impending changes to roles and responsibilities, and confirm that the individuals understand the changes and agree to follow them. Example 6: Implement and use tools and processes to promote communication and engagement among individuals with SDLC-related roles and responsibilities, such as creating messaging channels for team discussions. Example 7: Designate a group of individuals or a team as the code owner for each project.

Practices	Tasks	Notional Implementation Examples
	PO.2.2: Provide role-based training for all personnel with responsibilities that contribute to secure development. Periodically review personnel proficiency and role-based training, and update the training as needed.	Example 1: Document the desired outcomes of training for each role. Example 2: Define the type of training or curriculum required to achieve the desired outcome for each role. Example 3: Create a training plan for each role. Example 4: Acquire or create training for each role; acquired training may need to be customized for the organization. Example 5: Measure outcome performance to identify areas where changes to training may be beneficial.
	PO.2.3: Obtain upper management or authorizing official commitment to secure development, and convey that commitment to all with development-related roles and responsibilities.	Example 1: Appoint a single leader or leadership team to be responsible for the entire secure software development process, including being accountable for releasing software to production and delegating responsibilities as appropriate. Example 2: Increase authorizing officials' awareness of the risks of developing software without integrating security throughout the development life cycle and the risk mitigation provided by secure development practices. Example 3: Assist upper management in incorporating secure development support into their communications with personnel with development-related roles and responsibilities. Example 4: Educate all personnel with development-related roles and responsibilities on upper management's commitment to secure development and the importance of secure development to the organization.
Implement Supporting Toolchains (PO.3): Use automation to reduce human effort and improve the accuracy, reproducibility, usability, and comprehensiveness of security practices throughout the SDLC, as well as provide a way to document and demonstrate the use of these practices. Toolchains and tools may be used at different levels of the organization, such as organization-wide or project-specific, and may address a particular part of the SDLC, like a build pipeline.	**PO.3.1**: Specify which tools or tool types must or should be included in each toolchain to mitigate identified risks, as well as how the toolchain components are to be integrated with each other.	Example 1: Define categories of toolchains, and specify the mandatory tools or tool types to be used for each category. Example 2: Identify security tools to integrate into the developer toolchain. Example 3: Define what information is to be passed between tools and what data formats are to be used. Example 4: Evaluate tools' signing capabilities to create immutable records/logs for auditability within the toolchain. Example 5: Use automated technology for toolchain management and orchestration.

Practices	Tasks	Notional Implementation Examples
	PO.3.2: Follow recommended security practices to deploy, operate, and maintain tools and toolchains.	Example 1: Evaluate, select, and acquire tools, and assess the security of each tool. Example 2: Integrate tools with other tools and existing software development processes and workflows. Example 3: Use code-based configuration for toolchains (e.g., pipelines-as-code, toolchains-as-code). Example 4: Implement the technologies and processes needed for reproducible builds. Example 5: Update, upgrade, or replace tools as needed to address tool vulnerabilities or add new tool capabilities. Example 6: Continuously monitor tools and tool logs for potential operational and security issues, including policy violations and anomalous behavior. Example 7: Regularly verify the integrity and check the provenance of each tool to identify potential problems. Example 8: See PW.6 regarding compiler, interpreter, and build tools. Example 9: See PO.5 regarding implementing and maintaining secure environments.
	PO.3.3: Configure tools to generate artifacts of their support of secure software development practices as defined by the organization.	Example 1: Use existing tooling (e.g., workflow tracking, issue tracking, value stream mapping) to create an audit trail of the secure development-related actions that are performed for continuous improvement purposes. Example 2: Determine how often the collected information should be audited, and implement the necessary processes. Example 3: Establish and enforce security and retention policies for artifact data. Example 4: Assign responsibility for creating any needed artifacts that tools cannot generate.
Define and Use Criteria for Software Security Checks (PO.4): Help ensure that the software resulting from the SDLC meets the organization's expectations by defining and using criteria for checking the software's security during development.	**PO.4.1**: Define criteria for software security checks and track throughout the SDLC.	Example 1: Ensure that the criteria adequately indicate how effectively security risk is being managed. Example 2: Define key performance indicators (KPIs), key risk indicators (KRIs), vulnerability severity scores, and other measures for software security. Example 3: Add software security criteria to existing checks (e.g., the Definition of Done in agile SDLC methodologies). Example 4: Review the artifacts generated as part of the software development workflow system to determine if they meet the criteria. Example 5: Record security check approvals, rejections, and exception requests as part of the workflow and tracking system. Example 6: Analyze collected data in the context of the security successes and failures of each development project, and use the results to improve the SDLC.

Practices	Tasks	Notional Implementation Examples
	PO.4.2: Implement processes, mechanisms, etc. to gather and safeguard the necessary information in support of the criteria.	Example 1: Use the toolchain to automatically gather information that informs security decision-making. Example 2: Deploy additional tools if needed to support the generation and collection of information supporting the criteria. Example 3: Automate decision-making processes utilizing the criteria, and periodically review these processes. Example 4: Only allow authorized personnel to access the gathered information, and prevent any alteration or deletion of the information.
Implement and Maintain Secure Environments for Software Development (PO.5): Ensure that all components of the environments for software development are strongly protected from internal and external threats to prevent compromises of the environments or the software being developed or maintained within them. Examples of environments for software development include development, build, test, and distribution environments.	**PO.5.1**: Separate and protect each environment involved in software development.	Example 1: Use multi-factor, risk-based authentication and conditional access for each environment. Example 2: Use network segmentation and access controls to separate the environments from each other and from production environments, and to separate components from each other within each non-production environment, in order to reduce attack surfaces and attackers' lateral movement and privilege/access escalation. Example 3: Enforce authentication and tightly restrict connections entering and exiting each software development environment, including minimizing access to the internet to only what is necessary. Example 4: Minimize direct human access to toolchain systems, such as build services. Continuously monitor and audit all access attempts and all use of privileged access. Example 5: Minimize the use of production-environment software and services from non-production environments. Example 6: Regularly log, monitor, and audit trust relationships for authorization and access between the environments and between the components within each environment. Example 7: Continuously log and monitor operations and alerts across all components of the development environment to detect, respond, and recover from attempted and actual cyber incidents. Example 8: Configure security controls and other tools involved in separating and protecting the environments to generate artifacts for their activities. Example 9: Continuously monitor all software deployed in each environment for new vulnerabilities, and respond to vulnerabilities appropriately following a risk-based approach. Example 10: Configure and implement measures to secure the environments' hosting infrastructures following a zero trust architecture.

Practices	Tasks	Notional Implementation Examples
	PO.5.2: Secure and harden development endpoints (i.e., endpoints for software designers, developers, testers, builders, etc.) to perform development-related tasks using a risk-based approach.	Example 1: Configure each development endpoint based on approved hardening guides, checklists, etc.; for example, enable FIPS-compliant encryption of all sensitive data at rest and in transit. Example 2: Configure each development endpoint and the development resources to provide the least functionality needed by users and services and to enforce the principle of least privilege. Example 3: Continuously monitor the security posture of all development endpoints, including monitoring and auditing all use of privileged access. Example 4: Configure security controls and other tools involved in securing and hardening development endpoints to generate artifacts for their activities. Example 5: Require multi-factor authentication for all access to development endpoints and development resources. Example 6: Provide dedicated development endpoints on non-production networks for performing all development-related tasks. Provide separate endpoints on production networks for all other tasks. Example 7: Configure each development endpoint following a zero trust architecture.
Protect All Forms of Code from Unauthorized Access and Tampering (PS.1): Help prevent unauthorized changes to code, both inadvertent and intentional, which could circumvent or negate the intended security characteristics of the software. For code that is not intended to be publicly accessible, this helps prevent theft of the software and may make it more difficult or time-consuming for attackers to find vulnerabilities in the software.	**PS.1.1**: Store all forms of code – including source code, executable code, and configuration-as-code – based on the principle of least privilege so that only authorized personnel, tools, services, etc. have access.	Example 1: Store all source code and configuration-as-code in a code repository, and restrict access to it based on the nature of the code. For example, open-source code intended for public access may need its integrity and availability protected; other code may also need its confidentiality protected. Example 2: Use version control features of the repository to track all changes made to the code with accountability to the individual account. Example 3: Use commit signing for code repositories. Example 4: Have the code owner review and approve all changes made to the code by others. Example 5: Use code signing to help protect the integrity of executables. Example 6: Use cryptography (e.g., cryptographic hashes) to help protect file integrity.

Practices	Tasks	Notional Implementation Examples
Provide a Mechanism for Verifying Software Release Integrity (PS.2): Help software acquirers ensure that the software they acquire is legitimate and has not been tampered with.	**PS.2.1**: Make software integrity verification information available to software acquirers.	Example 1: Post cryptographic hashes for release files on a well-secured website. Example 2: Use an established certificate authority for code signing so that consumers' operating systems or other tools and services can confirm the validity of signatures before use. Example 3: Periodically review the code signing processes, including certificate renewal, rotation, revocation, and protection.
Archive and Protect Each Software Release (PS.3): Preserve software releases in order to help identify, analyze, and eliminate vulnerabilities discovered in the software after release.	**PS.3.1**: Securely archive the necessary files and supporting data (e.g., integrity verification information, provenance data) to be retained for each software release.	Example 1: Store the release files, associated images, etc. in repositories following the organization's established policy. Allow read-only access to them by necessary personnel and no access by anyone else. Example 2: Store and protect release integrity verification information and provenance data, such as by keeping it in a separate location from the release files or by signing the data.
	PS.3.2: Collect, safeguard, maintain, and share provenance data for all components of each software release (e.g., in a software bill of materials [SBOM]).	Example 1: Make the provenance data available to software acquirers in accordance with the organization's policies, preferably using standards-based formats. Example 2: Make the provenance data available to the organization's operations and response teams to aid them in mitigating software vulnerabilities. Example 3: Protect the integrity of provenance data, and provide a way for recipients to verify provenance data integrity. Example 4: Update the provenance data every time any of the software's components are updated.

Practices	Tasks	Notional Implementation Examples
Design Software to Meet Security Requirements and Mitigate Security Risks (PW.1): Identify and evaluate the security requirements for the software; determine what security risks the software is likely to face during operation and how the software's design and architecture should mitigate those risks; and justify any cases where risk-based analysis indicates that security requirements should be relaxed or waived. Addressing security requirements and risks during software design (secure by design) is key for improving software security and also helps improve development efficiency.	**PW.1.1**: Use forms of risk modeling – such as threat modeling, attack modeling, or attack surface mapping – to help assess the security risk for the software.	Example 1: Train the development team (security champions, in particular) or collaborate with a risk modeling expert to create models and analyze how to use a risk-based approach to communicate the risks and determine how to address them, including implementing mitigations. Example 2: Perform more rigorous assessments for high-risk areas, such as protecting sensitive data and safeguarding identification, authentication, and access control, including credential management. Example 3: Review vulnerability reports and statistics for previous software to inform the security risk assessment. Example 4: Use data classification methods to identify and characterize each type of data that the software will interact with.
	PW.1.2: Track and maintain the software's security requirements, risks, and design decisions.	Example 1: Record the response to each risk, including how mitigations are to be achieved and what the rationales are for any approved exceptions to the security requirements. Add any mitigations to the software's security requirements. Example 2: Maintain records of design decisions, risk responses, and approved exceptions that can be used for auditing and maintenance purposes throughout the rest of the software life cycle. Example 3: Periodically re-evaluate all approved exceptions to the security requirements, and implement changes as needed.

Practices	Tasks	Notional Implementation Examples
	PW.1.3: Where appropriate, build in support for using standardized security features and services (e.g., enabling software to integrate with existing log management, identity management, access control, and vulnerability management systems) instead of creating proprietary implementations of security features and services. [Formerly PW.4.3]	Example 1: Maintain one or more software repositories of modules for supporting standardized security features and services. Example 2: Determine secure configurations for modules for supporting standardized security features and services, and make these configurations available (e.g., as configuration-as-code) so developers can readily use them. Example 3: Define criteria for which security features and services must be supported by software to be developed.
Review the Software Design to Verify Compliance with Security Requirements and Risk Information (PW.2): Help ensure that the software will meet the security requirements and satisfactorily address the identified risk information.	**PW.2.1**: Have 1) a qualified person (or people) who were not involved with the design and/or 2) automated processes instantiated in the toolchain review the software design to confirm and enforce that it meets all of the security requirements and satisfactorily addresses the identified risk information.	Example 1: Review the software design to confirm that it addresses applicable security requirements. Example 2: Review the risk models created during software design to determine if they appear to adequately identify the risks. Example 3: Review the software design to confirm that it satisfactorily addresses the risks identified by the risk models. Example 4: Have the software's designer correct failures to meet the requirements. Example 5: Change the design and/or the risk response strategy if the security requirements cannot be met. Example 6: Record the findings of design reviews to serve as artifacts (e.g., in the software specification, in the issue tracking system, in the threat model).
Reuse Existing, Well-Secured Software When Feasible Instead of Duplicating Functionality (PW.4): Lower the costs of software development, expedite software development, and decrease the likelihood of introducing additional security vulnerabilities into the software by reusing software modules and services that have already had their security posture checked. This is particularly important for software that implements security functionality, such as cryptographic modules and protocols.	**PW.4.1**: Acquire and maintain well-secured software components (e.g., software libraries, modules, middleware, frameworks) from commercial, open-source, and other third-party developers for use by the organization's software.	Example 1: Review and evaluate third-party software components in the context of their expected use. If a component is to be used in a substantially different way in the future, perform the review and evaluation again with that new context in mind. Example 2: Determine secure configurations for software components, and make these available (e.g., as configuration-as-code) so developers can readily use the configurations. Example 3: Obtain provenance information (e.g., SBOM, source composition analysis, binary software composition analysis) for each software component, and analyze that information to better assess the risk that the component may introduce. Example 4: Establish one or more software repositories to host sanctioned and vetted open-source components. Example 5: Maintain a list of organization-approved commercial software components and component versions along with their provenance data. Example 6: Designate which components must be included in software to be developed. Example 7: Implement processes to update deployed software components to newer versions, and retain older versions of software components until all transitions from those versions have been completed successfully. Example 8: If the integrity or provenance of acquired binaries cannot be confirmed, build binaries from source code after verifying the source code's

Practices	Tasks	Notional Implementation Examples
	PW.4.2: Create and maintain well-secured software components in-house following SDLC processes to meet common internal software development needs that cannot be better met by third-party software components.	Example 1: Follow organization-established security practices for secure software development when creating and maintaining the components. Example 2: Determine secure configurations for software components, and make these available (e.g., as configuration-as-code) so developers can readily use the configurations. Example 3: Maintain one or more software repositories for these components. Example 4: Designate which components must be included in software to be developed. Example 5: Implement processes to update deployed software components to newer versions, and maintain older versions of software components until all transitions from those versions have been completed successfully.
	PW.4.4: Verify that acquired commercial, open-source, and all other third-party software components comply with the requirements, as defined by the organization, throughout their life cycles.	Example 1: Regularly check whether there are publicly known vulnerabilities in the software modules and services that vendors have not yet fixed. Example 2: Build into the toolchain automatic detection of known vulnerabilities in software components. Example 3: Use existing results from commercial services for vetting the software modules and services. Example 4: Ensure that each software component is still actively maintained and has not reached end of life; this should include new vulnerabilities found in the software being remediated. Example 5: Determine a plan of action for each software component that is no longer being maintained or will not be available in the near future. Example 6: Confirm the integrity of software components through digital signatures or other mechanisms. Example 7: Review, analyze, and/or test code. See PW.7 and PW.8.

Practices	Tasks	Notional Implementation Examples
Create Source Code by Adhering to Secure Coding Practices (PW.5): Decrease the number of security vulnerabilities in the software, and reduce costs by minimizing vulnerabilities introduced during source code creation that meet or exceed organization-defined vulnerability severity criteria.	**PW.5.1**: Follow all secure coding practices that are appropriate to the development languages and environment to meet the organization's requirements.	Example 1: Validate all inputs, and validate and properly encode all outputs. Example 2: Avoid using unsafe functions and calls. Example 3: Detect errors, and handle them gracefully. Example 4: Provide logging and tracing capabilities. Example 5: Use development environments with automated features that encourage or require the use of secure coding practices with just-in-time training-in-place. Example 6: Follow procedures for manually ensuring compliance with secure coding practices when automated methods are insufficient or unavailable. Example 7: Use tools (e.g., linters, formatters) to standardize the style and formatting of the source code. Example 8: Check for other vulnerabilities that are common to the development languages and environment. Example 9: Have the developer review their own human-readable code to complement (not replace) code review performed by other people or tools. See PW.7.
Configure the Compilation, Interpreter, and Build Processes to Improve Executable Security (PW.6): Decrease the number of security vulnerabilities in the software and reduce costs by eliminating vulnerabilities before testing occurs.	**PW.6.1**: Use compiler, interpreter, and build tools that offer features to improve executable security.	Example 1: Use up-to-date versions of compiler, interpreter, and build tools. Example 2: Follow change management processes when deploying or updating compiler, interpreter, and build tools, and audit all unexpected changes to tools. Example 3: Regularly validate the authenticity and integrity of compiler, interpreter, and build tools. See PO.3.
	PW.6.2: Determine which compiler, interpreter, and build tool features should be used and how each should be configured, then implement and use the approved configurations.	Example 1: Enable compiler features that produce warnings for poorly secured code during the compilation process. Example 2: Implement the "clean build" concept, where all compiler warnings are treated as errors and eliminated except those determined to be false positives or irrelevant. Example 3: Perform all builds in a dedicated, highly controlled build environment. Example 4: Enable compiler features that randomize or obfuscate execution characteristics, such as memory location usage, that would otherwise be predictable and thus potentially exploitable. Example 5: Test to ensure that the features are working as expected and are not inadvertently causing any operational issues or other problems. Example 6: Continuously verify that the approved configurations are being used. Example 7: Make the approved tool configurations available as configuration-as-code so developers can readily use them.

Practices	Tasks	Notional Implementation Examples
Review and/or Analyze Human-Readable Code to Identify Vulnerabilities and Verify Compliance with Security Requirements (PW.7): Help identify vulnerabilities so that they can be corrected before the software is released to prevent exploitation. Using automated methods lowers the effort and resources needed to detect vulnerabilities. Human-readable code includes source code, scripts, and any other form of code that an organization deems human-readable.	**PW.7.1**: Determine whether code review (a person looks directly at the code to find issues) and/or code analysis (tools are used to find issues in code, either in a fully automated way or in conjunction with a person) should be used, as defined by the organization.	Example 1: Follow the organization's policies or guidelines for when code review should be performed and how it should be conducted. This may include third-party code and reusable code modules written in-house. Example 2: Follow the organization's policies or guidelines for when code analysis should be performed and how it should be conducted. Example 3: Choose code review and/or analysis methods based on the stage of the software.
	PW.7.2: Perform the code review and/or code analysis based on the organization's secure coding standards, and record and triage all discovered issues and recommended remediations in the development team's workflow or issue tracking system.	Example 1: Perform peer review of code, and review any existing code review, analysis, or testing results as part of the peer review. Example 2: Use expert reviewers to check code for backdoors and other malicious content. Example 3: Use peer reviewing tools that facilitate the peer review process, and document all discussions and other feedback. Example 4: Use a static analysis tool to automatically check code for vulnerabilities and compliance with the organization's secure coding standards with a human reviewing the issues reported by the tool and remediating them as necessary. Example 5: Use review checklists to verify that the code complies with the requirements. Example 6: Use automated tools to identify and remediate documented and verified unsafe software practices on a continuous basis as human-readable code is checked into the code repository. Example 7: Identify and document the root causes of discovered issues. Example 8: Document lessons learned from code review and analysis in a wiki that developers can access and search.

Practices	Tasks	Notional Implementation Examples
Test Executable Code to Identify Vulnerabilities and Verify Compliance with Security Requirements (PW.8): Help identify vulnerabilities so that they can be corrected before the software is released in order to prevent exploitation. Using automated methods lowers the effort and resources needed to detect vulnerabilities and improves traceability and repeatability. Executable code includes binaries, directly executed bytecode and source code, and any other form of code that an organization deems executable.	**PW.8.1**: Determine whether executable code testing should be performed to find vulnerabilities not identified by previous reviews, analysis, or testing and, if so, which types of testing should be used.	Example 1: Follow the organization's policies or guidelines for when code testing should be performed and how it should be conducted (e.g., within a sandboxed environment). This may include third-party executable code and reusable executable code modules written in-house. Example 2: Choose testing methods based on the stage of the software.
	PW.8.2: Scope the testing, design the tests, perform the testing, and document the results, including recording and triaging all discovered issues and recommended remediations in the development team's workflow or issue tracking system.	Example 1: Perform robust functional testing of security features. Example 2: Integrate dynamic vulnerability testing into the project's automated test suite. Example 3: Incorporate tests for previously reported vulnerabilities into the project's test suite to ensure that errors are not reintroduced. Example 4: Take into consideration the infrastructures and technology stacks that the software will be used with in production when developing test plans. Example 5: Use fuzz testing tools to find issues with input handling. Example 6: If resources are available, use penetration testing to simulate how an attacker might attempt to compromise the software in high-risk scenarios. Example 7: Identify and record the root causes of discovered issues. Example 8: Document lessons learned from code testing in a wiki that developers can access and search. Example 9: Use source code, design records, and other resources when developing test plans.

Practices	Tasks	Notional Implementation Examples
Configure Software to Have Secure Settings by Default (PW.9): Help improve the security of the software at the time of installation to reduce the likelihood of the software being deployed with weak security settings, putting it at greater risk of compromise.	**PW.9.1**: Define a secure baseline by determining how to configure each setting that has an effect on security or a security-related setting so that the default settings are secure and do not weaken the security functions provided by the platform, network infrastructure, or services.	Example 1: Conduct testing to ensure that the settings, including the default settings, are working as expected and are not inadvertently causing any security weaknesses, operational issues, or other problems.
	PW.9.2: Implement the default settings (or groups of default settings, if applicable), and document each setting for software administrators.	Example 1: Verify that the approved configuration is in place for the software. Example 2: Document each setting's purpose, options, default value, security relevance, potential operational impact, and relationships with other settings. Example 3: Use authoritative programmatic technical mechanisms to record how each setting can be implemented and assessed by software administrators. Example 4: Store the default configuration in a usable format and follow change control practices for modifying it (e.g., configuration-as-code).
Identify and Confirm Vulnerabilities on an Ongoing Basis (RV.1): Help ensure that vulnerabilities are identified more quickly so that they can be remediated more quickly in accordance with risk, reducing the window of opportunity for attackers.	**RV.1.1**: Gather information from software acquirers, users, and public sources on potential vulnerabilities in the software and third-party components that the software uses, and investigate all credible reports.	Example 1: Monitor vulnerability databases , security mailing lists, and other sources of vulnerability reports through manual or automated means. Example 2: Use threat intelligence sources to better understand how vulnerabilities in general are being exploited. Example 3: Automatically review provenance and software composition data for all software components to identify any new vulnerabilities they have.

Practices	Tasks	Notional Implementation Examples
	RV.1.2: Review, analyze, and/or test the software's code to identify or confirm the presence of previously undetected vulnerabilities.	Example 1: Configure the toolchain to perform automated code analysis and testing on a regular or continuous basis for all supported releases. Example 2: See PW.7 and PW.8.
	RV.1.3: Have a policy that addresses vulnerability disclosure and remediation, and implement the roles, responsibilities, and processes needed to support that policy.	Example 1: Establish a vulnerability disclosure program, and make it easy for security researchers to learn about your program and report possible vulnerabilities. Example 2: Have a Product Security Incident Response Team (PSIRT) and processes in place to handle the responses to vulnerability reports and incidents, including communications plans for all stakeholders. Example 3: Have a security response playbook to handle a generic reported vulnerability, a report of zero-days, a vulnerability being exploited in the wild, and a major ongoing incident involving multiple parties and open-source software components. Example 4: Periodically conduct exercises of the product security incident response processes.
Assess, Prioritize, and Remediate Vulnerabilities (RV.2): Help ensure that vulnerabilities are remediated in accordance with risk to reduce the window of opportunity for attackers.	**RV.2.1**: Analyze each vulnerability to gather sufficient information about risk to plan its remediation or other risk response.	Example 1: Use existing issue tracking software to record each vulnerability. Example 2: Perform risk calculations for each vulnerability based on estimates of its exploitability, the potential impact if exploited, and any other relevant characteristics.

Practices	Tasks	Notional Implementation Examples
	RV.2.2: Plan and implement risk responses for vulnerabilities.	Example 1: Make a risk-based decision as to whether each vulnerability will be remediated or if the risk will be addressed through other means (e.g., risk acceptance, risk transference), and prioritize any actions to be taken. Example 2: If a permanent mitigation for a vulnerability is not yet available, determine how the vulnerability can be temporarily mitigated until the permanent solution is available, and add that temporary remediation to the plan. Example 3: Develop and release security advisories that provide the necessary information to software acquirers, including descriptions of what the vulnerabilities are, how to find instances of the vulnerable software, and how to address them (e.g., where to get patches and what the patches change in the software; what configuration settings may need to be changed; how temporary workarounds could be implemented). Example 4: Deliver remediations to acquirers via an automated and trusted delivery mechanism. A single remediation could address multiple vulnerabilities. Example 5: Update records of design decisions, risk responses, and approved exceptions as needed. See PW.1.2.
Analyze Vulnerabilities to Identify Their Root Causes (RV.3): Help reduce the frequency of vulnerabilities in the future.	**RV.3.1**: Analyze identified vulnerabilities to determine their root causes.	Example 1: Record the root cause of discovered issues. Example 2: Record lessons learned through root cause analysis in a wiki that developers can access and search.
	RV.3.2: Analyze the root causes over time to identify patterns, such as a particular secure coding practice not being followed consistently.	Example 1: Record lessons learned through root cause analysis in a wiki that developers can access and search. Example 2: Add mechanisms to the toolchain to automatically detect future instances of the root cause. Example 3: Update manual processes to detect future instances of the root cause.
	RV.3.3: Review the software for similar vulnerabilities to eradicate a class of vulnerabilities, and proactively fix them rather than waiting for external reports.	Example 1: See PW.7 and PW.8.

Practices	Tasks	Notional Implementation Examples
	RV.3.4: Review the SDLC process, and update it if appropriate to prevent (or reduce the likelihood of) the root cause recurring in updates to the software or in new software that is created.	Example 1: Record lessons learned through root cause analysis in a wiki that developers can access and search. Example 2: Plan and implement changes to the appropriate SDLC practices.