Carnegie Mellon University
Software Engineering Institute

# DoD Developer's Guidebook for Software Assurance

Dr. William R. Nichols, Jr.
Dr. Thomas Scanlon

**December 2018**

**SPECIAL REPORT**
CMU/SEI-2018-SR-013

**Software Solutions and CERT Divisions**



http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.



This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100



DM18-1005

# Table of Contents

# List of Figures

# List of Tables

# Executive Summary

Software assurance refers to the justified confidence that software functions as intended and is free of vulnerabilities throughout the product lifecycle. While "free of vulnerabilities" is the ideal, in practice the objective is to manage the risk associated with vulnerabilities. This guidebook helps software developers understand expectations for software assurance and provides guidance for selecting and applying software security tools and techniques, which are rapidly growing in number, to manage that risk. Because developers also need to be aware of the regulatory background in which their projects operate, this guidebook also summarizes many of the standards and requirements that affect software assurance decisions.

This guidebook provides a broad focus because security must be maintained throughout all phases of the product and development lifecycle. While developers are mostly concerned with development and maintenance, they require a basic awareness of all processes for several reasons. Software requirements and software architecture place many constraints on the development. Many products include commercial off-the-shelf, government off-the-shelf, or open-source software components, so developers must be aware of risks introduced through the acquisition and supply chain. Transition increasingly includes monitoring the product in use to identify threats, and verification and validation are needed to consider failure cases.

A large number of publications and resources provide valuable information for developers, making it difficult to collect and summarize them all in one place. Instead, this guidebook provides pointers to key resources that developers should consult when appropriate. Because the *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* report [Wheeler 2016] is particularly valuable for developers creating software for the Department of Defense (DoD), we have included a summary of the report and its approach for selecting tools in this guidebook.

The tool selection process cannot be reduced to a simple flowchart or algorithm because there are so many interacting factors. This guidebook provides a bottom-up approach to tool selection, considering what activities and tools are normally appropriate at different stages of the development or product lifecycle. It also includes guidance for special lifecycle considerations, such as new development and system reengineering.

Metrics that may be useful in selecting and applying tools or techniques during development are also discussed. After tools are integrated into the environment, the tool findings must be addressed. To do so developers need to be capable of communicating the costs and risks to program management. This guidebook describes the costs and benefit decisions relevant to developers so they will be aware of what management needs to know and how to communicate it appropriately.

Special sections are devoted to assurance in software sustainment and software acquisition. During sustainment—once products are in use—they should be monitored, and the evolving threats should be modeled. Software acquisition is a special case in which most or all of the product is developed by third parties, which requires special considerations for managing risk.

Guidance for both situations is provided. Finally, the appendices provide references, definitions, and tools to support software assurance decisions.

# Abstract

Software assurance refers to the justified confidence that software functions as intended and is free of vulnerabilities throughout the product lifecycle. While "free of vulnerabilities" is the ideal, in practice the objective is to manage the risk associated with vulnerabilities. To that end, this guidebook helps software developers understand expectations for software assurance. Because developers need to be aware of the regulatory background in which their projects operate, this guidebook summarizes standards and requirements that affect software assurance decisions and provides pointers to key resources that developers should consult. It includes a summary of the *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* report, along with its approach for selecting tools. A bottom-up approach to tool selection is also provided, which considers what activities and tools are typically appropriate at different stages of the development or product lifecycle. Advice is provided for special lifecycle considerations, such as new development and system reengineering, and metrics that may be useful in selecting and applying tools or techniques during development are discussed. Special sections are devoted to assurance in software sustainment and software acquisition. Supplemental materials are provided in the appendices.

# 1 Introduction

As threats to software systems grow, so do the security tools, requirements, and regulations used to combat them. The work to secure software falls upon software developers. This guidebook helps software developers understand software assurance requirements and provides guidance for applying the growing body of software security tools and techniques. This guidebook is not intended to be a tutorial or training for secure development. This guidebook should be used for general knowledge and principles of software assurance, and as a starting point for finding more detailed resources.

## 1.1 Using This Guidebook

Sections 1 and 2 of this guidebook provide general information about assurance, with Section 2 describing software assurance principles in the context of the lifecycle and development processes.

Section 3 provides an overview of a minimal set of security activities in the different lifecycle processes. Use this section as a first step in constructing a secure development workflow. Section 4 provides information on measuring the process. Section 5 provides a short guide to using the *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* report, which is the comprehensive source for software assurance tools and techniques. The remainder of the guidebook provides more detailed instructions on selecting tools and creating a secure workflow, with instructions for special circumstances, such as during sustainment and acquisition.

Software developers must also be aware of the regulatory background in which their projects operate, so this guidebook also summarizes many of the laws and requirements that affect software assurance decisions. Throughout this guidebook we provide pointers to additional software assurance resources that expand on the content we have covered. A collection of all these resources appears in Appendix B.

## 1.2 Defining Software Assurance

The National Defense Industrial Association provides the following definition in *Engineering for System Assurance*:

> *System assurance (SA) is the justified confidence that the system functions as intended and is free of exploitable vulnerabilities, either intentionally or unintentionally designed or inserted as part of the system at any time during the life cycle. This ideal of no exploitable vulnerabilities is usually unachievable in practice, so programs must perform risk management to reduce the probability and impact of vulnerabilities to acceptable levels* [NDIA 2008].

The ideal software system is free from vulnerabilities, and the level of confidence in this target is often used as a definition of software assurance. A more practical definition emphasizes risk management by balancing cost and potential loss. The following risk-based interpretation is

provided by the Carnegie Mellon University Software Engineering Institute (SEI) in *Predicting Software Assurance Using Quality and Reliability Measures:*

> *the level of confidence we have that a system behaves as expected and the security risks associated with the business use of the software are acceptable* [Woody 2014]

## 1.3 DoD Software Assurance Requirements

There are statutory requirements, regulatory requirements, and guidelines for software developed by and for the DoD. The list below highlights the DoD requirements specific to software assurance as discussed in this guidebook. See Appendix A for a more extensive discussion of regulatory requirements.

- Software Assurance Plan (NDAA for Fiscal Year 2013, Section 932, Improvements in Assurance of Computer Software Procured by the Department of Defense, January 2, 2013)
- Program Protection Plan DoDI 5000.02 [USD(AT&L) 2017] and DoDI 5200.39 [DoD 2011]
- Risk Management Framework, which has replaced the DoD Information Assurance Certification and Accreditation Process (DIACAP) requirement [NIST 2010]
- Improvements in assurance of computer software procured by the DoD (NDAA for Fiscal Year 2013)

## 1.4 Software Assurance Resources

The resources listed below provide broad information about the subject of software assurance and are important assets for DoD developers.

*Table 1: Resource List for Assurance Information*

| Resource | Description |
|---|---|
| *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* [Wheeler 2016] | A publication by the Institute for Defense Analyses (IDA) that contains a large volume of information on the types of tools available and contextual factors on how they can affect security. |
| *Building Security In Maturity Model (BSIMM)* [McGraw 2017] | A study of existing software security initiatives sponsored by the Department of Homeland Security. It collects the state of professional practice, but does not recommend specific practices. |
| *Cyber Security Engineering: A Practical Approach* [Mead 2016] | A book in the SEI Series on Software Engineering. This publication provides a reference and tutorial on a broad range of assurance issues and practices. |
| SAFECode (https://safecode.org) | An industry group “dedicated to increasing trust in information and communications technology products and services through the advancement of effective software assurance methods.” |
| Intellipedia at Intelink (https://intellipedia.intellink.gov) | A collection of wikis available to individuals with appropriate clearances. These online resources contain information on various software assurance topics relevant to DoD developers and contractors. |

# 2 Software Assurance Concepts

Software can be subjected to a variety of attacks depending on its domain and use. The software must be secured, and software developers are responsible for taking appropriate steps during all phases of the product lifecycle to assure that the software has adequate resistance to attacks. The requirements that must be met are regulatory, legal, and contractual, and all are directed to assure the software product in use. Justified confidence that the software complies requires evidence that software security activities were not only performed but were performed effectively.

## 2.1 Overview of Security Attributes and Exploits

As a quality attribute of software-intensive systems, security must be engineered architecturally into the system and be explicitly considered during all lifecycle stages and processes.

The key requirements of security are confidentiality, integrity, and availability (CIA). Many organizations also include authentication and non-repudiation. These requirements are described below:

- confidentiality – the ability to prevent exfiltration of data in a system, keeping proprietary, sensitive, or personal information private and inaccessible to those not authorized
- integrity – the maintenance of authenticity, accuracy, and completeness of the program
- availability – the ability to continue to provide the data or service as required
- authentication – a security measure designed to establish the validity of a transmission, message, or originator; or a means of verifying an individual's authorization to receive specific categories of information
- non-repudiation – a key DoD requirement, a process in which the sender of data is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the data

Exploits typically attempt one of the following [Alberts 2003]:

- disclosure of data (violation of the confidentiality attribute)
- modification of data (violation of the integrity attribute)
- insertion of false data (violation of the integrity attribute)
- destruction of data (violation of the availability attribute)
- interruption of access to data (violation of the availability attribute)
- system destruction, destabilization, or degradation (violation of the availability attribute)

## 2.2 Principles of Software Assurance

Developing effectively secure systems requires a combination of approaches to ensure mission success, including minimizing vulnerabilities and managing the remaining vulnerabilities. These approaches should be driven by threat analysis and oriented to mission success—in other words, they should address the prioritized threats that will most impact mission success.

In *The Protection of Information in Computer Systems* [Saltzer 1975], the authors proposed basic software design principles that focus on protection mechanisms to "guide the design and contribute to an implementation without security flaws." These principles include the following:

- Economy of mechanism – Keep the design as simple and small as possible.
- Fail-safe defaults – Base access decisions on permission rather than exclusion.
- Complete mediation – Check every access to every object for authority.
- Open design – Do not keep the design secret. The mechanisms should not depend on the ignorance of potential attackers but rather on the possession of specific, more easily protected, keys or passwords.
- Separation of privilege – Where feasible, use a protection mechanism that requires two keys to unlock it. This design is more robust and flexible than one that allows access to the presenter of only a single key.
- Least privilege – Every program and every user of the system should operate using the least set of privileges necessary to complete the job.
- Least common mechanism – Minimize the number of mechanisms common to more than one user and depended on by all users.
- Psychological acceptability – Design the human interface for ease of use so that users routinely and automatically apply the protection mechanisms correctly.

These principles are still useful today, but Mead and Woody recommend extending them with the following principles [Mead 2016]:

- Risk shall be properly understood to drive appropriate assurance decisions.
- Risk concerns shall be aligned across all stakeholders and all interconnected technology elements.
- Dependencies shall not be trusted until proven trustworthy.
- Attacks shall be expected.
- Assurance requires effective coordination among all technology participants.
- Assurance shall be well planned and dynamic.
- A means to measure and audit overall assurance shall be built in.

Another related principle is to assume all connections and input are untrusted by default (e.g., white-listing).

These are all useful principles, but they do not tell developers specifically what they should do or when they should do it. This is because principles are general and context-free advice. Specific actions and practices must be adopted, but the specifics depend on context.

When we say that "assurance shall be built in," the principles must be realized in practice. "Building in" is sometimes called "shifting to the left" in the development lifecycle by explicitly applying security practices early during acquisition, requirements, design, and development. How the developers can select practices appropriate to their context is a key subject of much of this guidebook.

## 2.3 Lifecycle Assurance

The consequences of assurance are realized during software operation. However, the actions of anticipating, preventing, responding, mitigating, and remediating take place throughout the product lifecycle. This section summarizes some lifecycle considerations that are addressed in more detail in later sections.

### 2.3.1 Lifecycle Stages and Processes

The lifecycle stages most commonly used by the DoD are shown in Figure 1. These stages are used for accounting during the system development, use, and retirement.

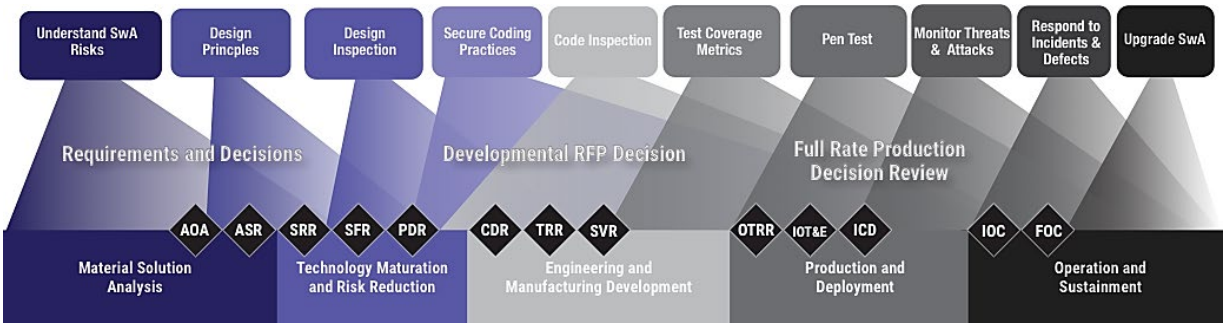



*Figure 1: DoD Lifecycle Stages*

The processes used in software development are described in ISO 15288 [ISO/IEC/IEEE 2015] and 12207 [ISO/IEC 2008]. ISO 15288, for example, describes processes that are used during the development of systems, such as

6.4.1 - Stakeholder Requirements Definition Process

6.4.2 - Requirements Analysis Process

6.4.3 - Architectural Design Process

6.4.4 - Implementation Process

6.4.5 - Integration Process

6.4.6 - Verification Process

6.4.7 - Transition Process

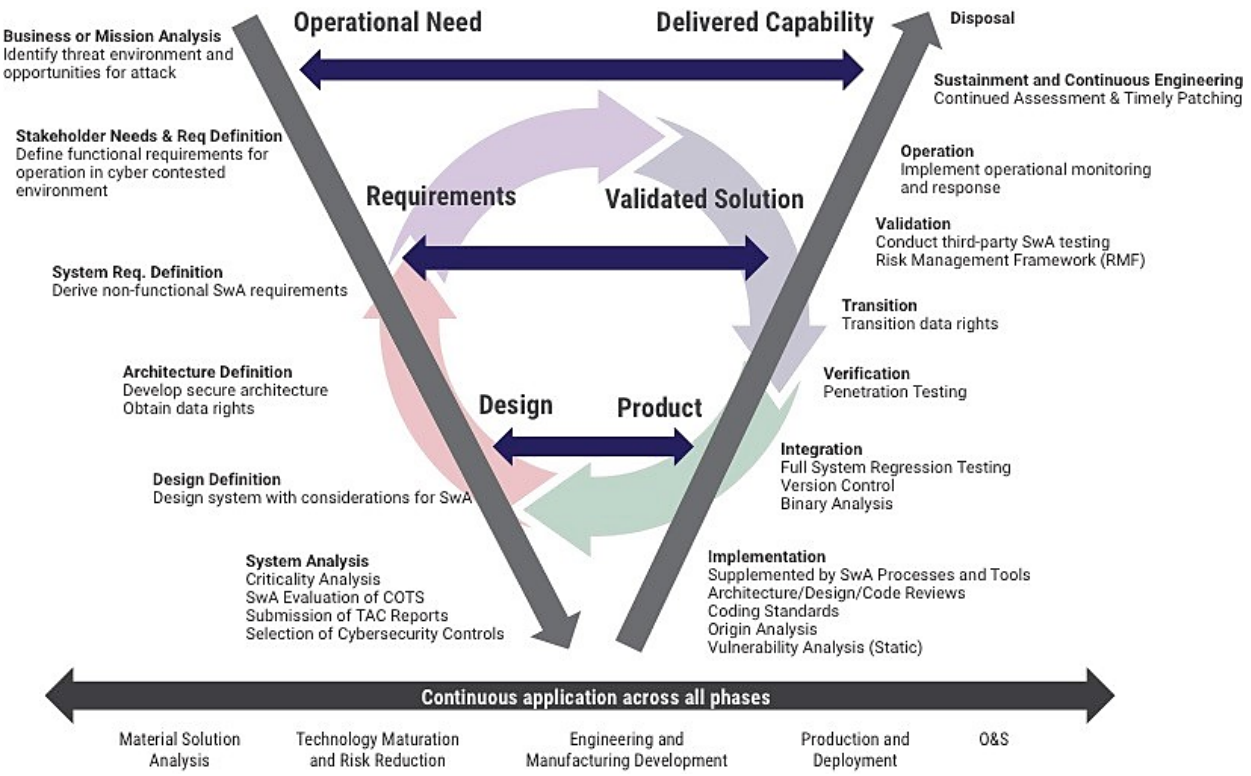
6.4.8 - Validation Process

6.4.9 - Operation Process

6.4.10 - Maintenance Process

6.4.11 - Disposal Process

While lifecycle stage refers to the product maturity, the processes refer to types of activities performed on the product, and practices refer to the specific activities performed. Some processes and practices are more common during some stages than others. For example, secure coding becomes relevant in the stages of development through operations, while test coverage becomes relevant pre-milestone B. Some of the secure practices performed during lifecycle processes are shown in Figure 2.




derived with permission from a diagram originally developed by the Defense Acquisition University [Butler 2016]

*Figure 2: Software Assurance Practices Applied Throughout the Development Lifecycle*

An alternate mapping of the processes can be found in ISO/IEC 12207 [ISO/IEC 2008], which also describes software lifecycle processes. The primary 12207 categories of processes include the following:

- acquisition
- supply
- development
- operation
- maintenance
- destruction

This grouping of processes appears to map processes to a predominant stage and can lead to confusion between the stage and the processes. In practice, the development processes and practices occur in parallel, or in repeated cycles, as depicted in Figure 3. Development thus includes requirements through validation processes. Moreover, the development processes are also used to perform maintenance tasks. While viewing the many different mappings that exist can be confusing, Figure 3 is included here because it is common enough that developers need to be aware of it.

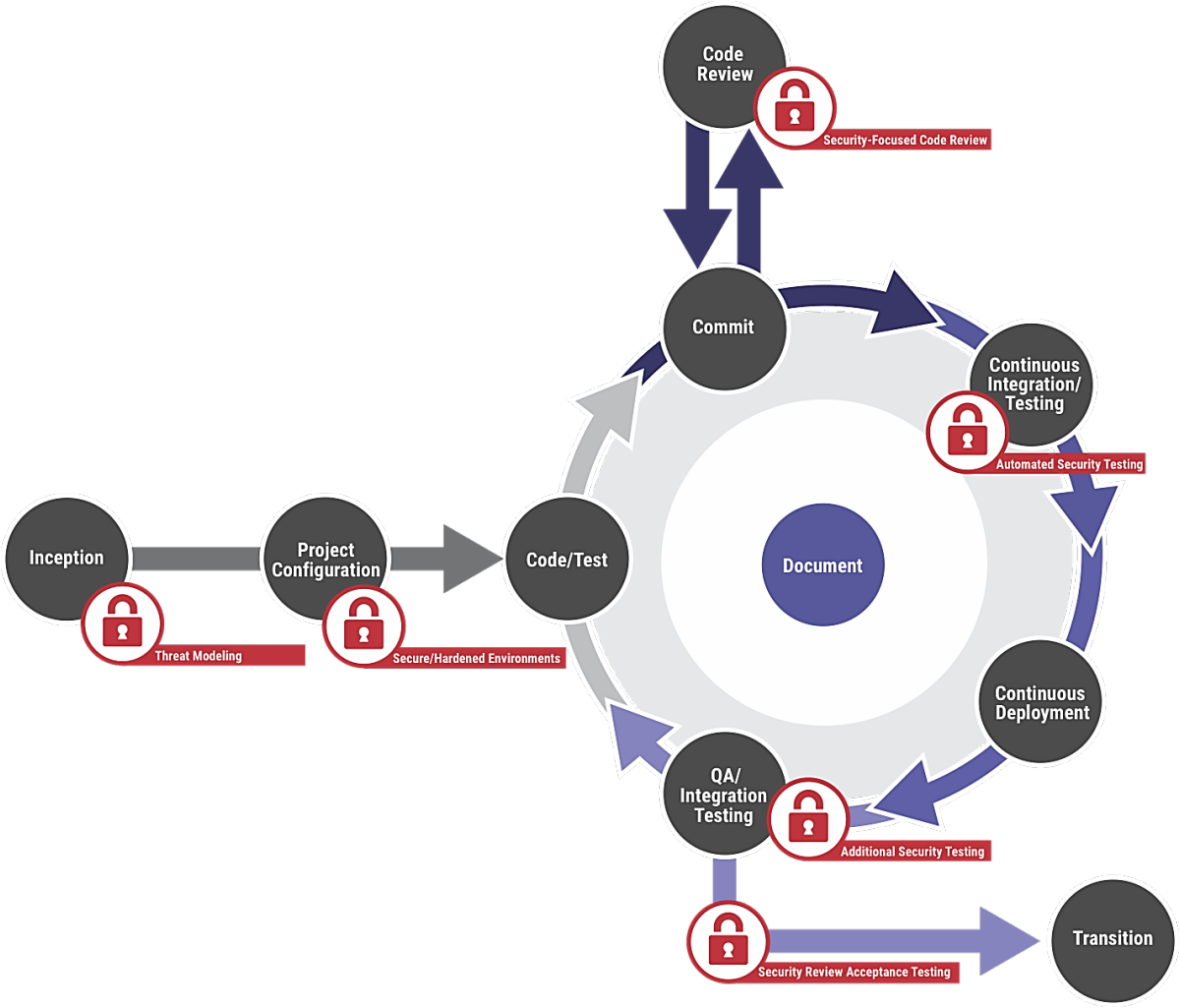



*Figure 3: Cyclic View of the Software Development Cycle*

While developers are mostly concerned with development and maintenance, they require a basic awareness of all processes for several reasons. The decisions made during requirements and architectural design often place global constraints on later development activities. For example, because many products include commercial off-the-shelf (COTS), government off-the-shelf (GOTS), or open-source software (OSS) components, risks are introduced through the supply chain that developers must be aware of. Transition increasingly includes monitoring the product in use to identify threats, and verification and validation are needed to consider failure cases.

### 2.3.2 Lifecycle Assurance Resources

*Table 2: Resource List for Lifecycle Assurance*

| Resource | Description |
|---|---|
| *The DoD Program Manager's Guide to Software Assurance* | An SEI document that is a companion to this guidebook |
| *Guide for Applying the Risk Management Framework to Federal Information Systems* [NIST 2010] | Guidelines published by NIST for applying the Risk Management Framework to federal information systems |
| *State-of-the-Art-Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* [Wheeler 2016] | A publication by the IDA that contains a large volume of information on the types of tools available and contextual factors on how they can affect security |
| *Building Security In Maturity Model (BSIMM)* [McGraw 2017] | A study of existing software security initiatives sponsored by the Department of Homeland Security. It collects the state of professional practice but does not recommend specific practices. |

## 2.4 Secure Practices Across the Lifecycle

The CERT secure coding wiki maintains a list of practices[1] that are summarized and reorganized in the list below. Although these techniques are called "secure coding," only a few are actually specific to the coding phase. The practices involve many lifecycle phases, including requirements, design and code, verification, and validation.

1. Model threats. Use threat modeling and develop threat mitigation strategies that are implemented in design, code, and test cases [Swiderski 2009].
2. Define security requirements. Identify and document security requirements early in the development lifecycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements.
3. Architect and design for security policies. Create a software architecture, and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
4. Keep it simple. Keep the design as simple and small as possible [Saltzer 1974, 1975]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use.
5. Default deny. Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted [Saltzer 1974, 1975].
6. Adhere to the principle of least privilege. Every process should execute with the least set of privileges necessary to complete the job [Saltzer 1974, 1975].
7. Sanitize data sent to other systems. Sanitize all data passed to complex subsystems such as command shells, relational databases, and (COTS) components.

[1] See https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices

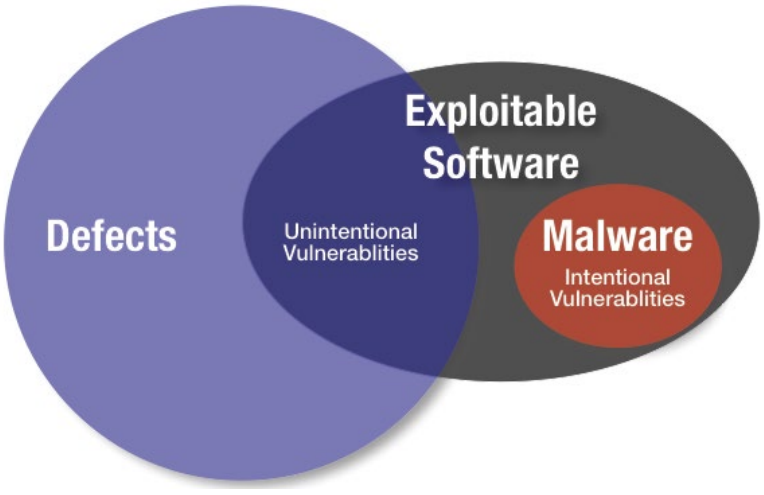
8. Practice defense in depth. Manage risk with multiple defensive strategies so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability, limit the consequences of a successful exploit, or both.
9. Validate input. Validate input from all untrusted data sources. The default source condition should be untrusted, for example, by using a whitelist rather than a blacklist.
10. Adopt a secure coding standard.
11. Heed compiler warnings. Compile code using the highest warning level available for your compiler, and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
12. Use effective quality assurance techniques. Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems.

### 2.4.1 Lifecycle Costs for Software Assurance

"A rising tide lifts all boats" is an aphorism that can be applied to how software quality affects security. Although the relationship between quality and security is not fully defined, expert opinion is that defective software is not secure. This notion is supported by empirical data that a significant portion of software defects is also a weakness that can potentially be exploited [Woody 2015]. The terms *weakness* and *vulnerability* can have many meanings. In this guidebook, we use the operational definitions from MITRE:

> *Software weaknesses are errors that can lead to software vulnerabilities. Software vulnerabilities, such as those enumerated on the Common Vulnerabilities and Exposures (CVE) List, are mistakes in software that can be directly used by a hacker to gain access to a system or network* [MITRE 2017].

The relationship can be notionally visualized in Figure 4.




adapted with permission from the DASD(SE)

*Figure 4: Example of Overlapping Vulnerabilities and Defects*

Overall quality is a necessary, but not fully sufficient, condition for security. Poor overall quality undermines software product security. Yet a general lack of faults and defects is not a sufficient indicator because not all aspects of security are addressed with normal defect removal activities. Security aspects such as confidentiality, integrity, and availability must also be considered in the requirements and included in specific design goals.

Some general quality guidelines are provided below.

- Quality activities should improve cost and schedule performance rather than being considered net cost.
- Prevention techniques should be used prior to removal techniques to improve security and quality.
- Removal techniques should be applied as early in the development as practicable. Because no removal technique finds all defects, applying a variety of techniques is most effective.
- Tests should verify the product vulnerability risk level as the primary means to identify and remove vulnerabilities. Tests are necessary, but not sufficient, to assure quality.

SEI researchers provide quantitative empirical data showing that a substantial portion of weaknesses can be removed with common quality techniques [Woody 2015].

To determine this, SEI researchers first examined the Common Weakness Enumeration (CWE) top 25 [MITRE 2018b], and approximately 50% were found to be removable with standard quality techniques. Second, a set of low-defect products were examined and found to have very low overall weaknesses and safety-critical densities (see Figure 5). Third, a review of literature that discussed the ratio of known vulnerabilities to overall defects found almost half of the sample fell within the 1% to 5% range (see Figure 6). From these observations, we conclude that applying overall quality techniques throughout the lifecycle should be effective for removing a substantial portion of security vulnerabilities.

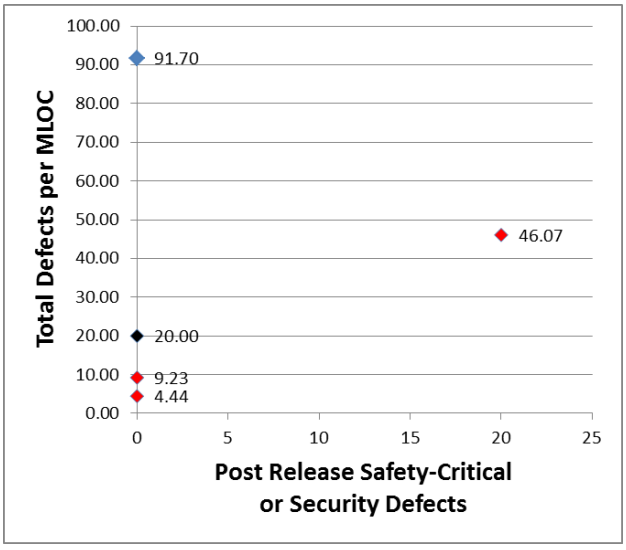



*Figure 5: Security and Safety-Critical Defect Density vs. Overall Defect Density*

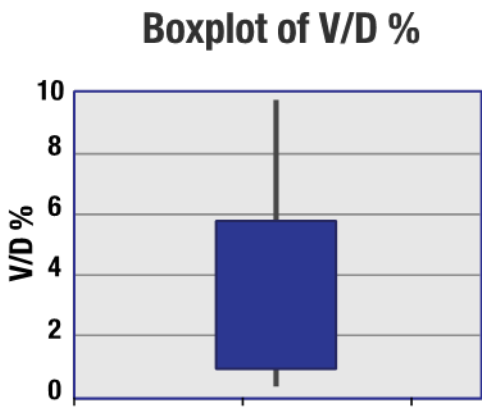



*Figure 6: Ratios of Vulnerability Density to Overall Defect Density*

Weakness can be thought of as a type of defect. Some weaknesses are at the implementation level (e.g., buffer overflows) and others are related to design or requirements (e.g., failure to require authentication). Every activity in which software is constructed, starting with requirements, includes the creation of errors, mistakes, defects, and vulnerabilities. Some of these are discovered and corrected immediately, but some require additional steps, including, but not limited to, compiler flags, static analysis, virus checks, or penetration tests.

In the development process, defect injection has been modeled using tanks (into which vulnerabilities are injected as product is created) and filters (which remove defects as the product flows to downstream phases). An example is shown in Figure 7. Each tank is a construction activity, and each filter is a removal activity. This guidebook helps a developer or acquirer determine a reasonable set of filters (i.e., security tools and techniques) and where they should be placed in the lifecycle workflow. Refer to Figure 2 for an example of software assurance practices applied throughout the development lifecycle.

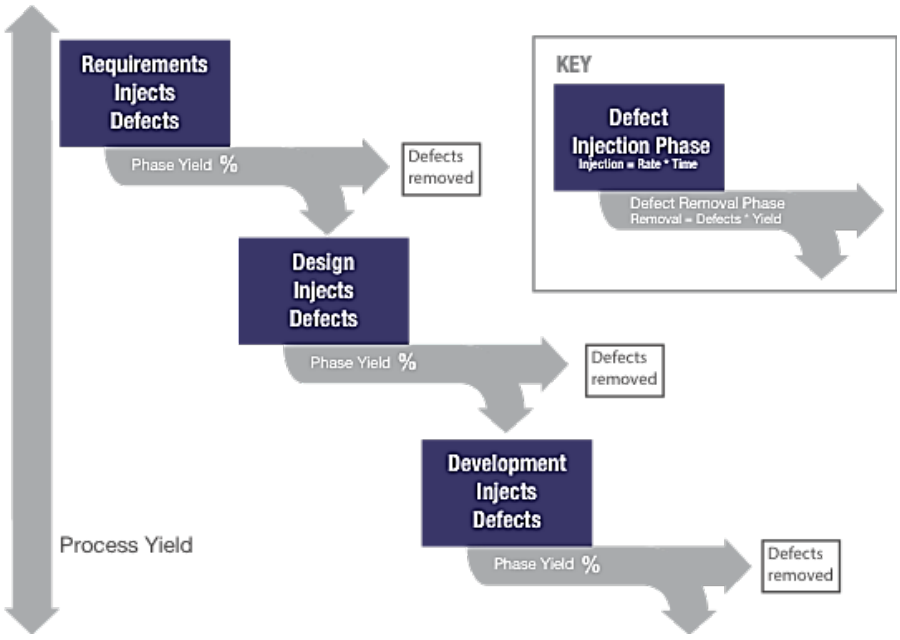



*Figure 7: Tank and Filter Injection and Removal Mode*

As software development workflows are composed, consider the overall defect control benefits of tools and techniques to both security and total cost. Data collected by Capers Jones [Jones 2009] shows that the number of hours required to repair code defects increases substantially the later you are in the development cycle:

- 15 minutes to fix an issue at implementation
- 1 hour to fix at integration build
- 12.5 hours to fix at test
- 25 hours to fix at production

Figure 8 shows how defect find-and-fix time increases during later development activities.

Jones shows how higher quality reduces total cost of ownership by lowering the costs of defects in production [Jones 2009]. These costs do not include lost productivity during patch deployment or the economic damage resulting from security breaches. In summary, higher levels of software assurance are warranted when a more complete accounting of the costs is done.

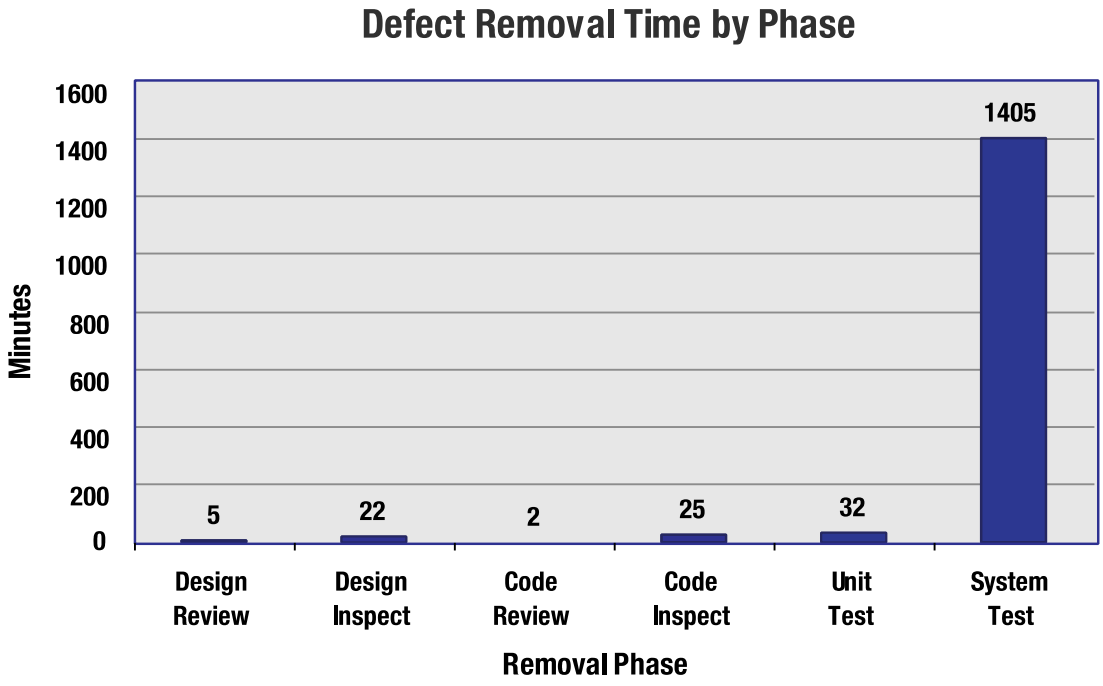



*Figure 8: Total Cost of Defect Removal Across Development Phases*

# 3 Quick-Start Guide to Assurance, by Lifecycle Phase

Knowing what tools, techniques, and countermeasures are available during each development process is a useful starting point for selecting cost-effective security assurance activities. Nonetheless, the development activity provides only a single view of a multidimensional problem. The *SOAR*, summarized in Section 5, provides more insight into the other considerations. A more complete selection process is summarized in Section 6. Table 15 summarizes the lifecycle processes where tools and techniques can be used.

While composing a cost-effective set of software assurance activities requires more than selecting a set of available tools, professionals need to know not only which tools can be used but also which should *almost always* be used. Moreover, the guidelines below must be followed.

- Developers must know how to use the tools effectively, which requires training.
- The tools must be used properly and consistently, which requires planning, discipline, and often automation.
- Evidence that software security assurance activities were performed along with some measure of effectiveness must be collected to provide project management with sufficient confidence that the tools or techniques are effective and that the project complies with regulatory requirements.

This section describes tools, techniques, and countermeasures assuming the iterative and incremental development lifecycle viewpoint of the developer as depicted in Figure 2 and the lifecycle processes listed in Section 2.3.1.

## 3.1 Stakeholder Requirements Definition

This step precedes the technical objectives. However, the requirements and their analysis will be used later when setting and prioritizing the technical objectives for security.

There are currently few tools specifically dedicated to eliciting and documenting security requirements. Of course, general-purpose requirements management tools can and should be used. For security-specific requirements, we recommend the following, in order of accessibility:

- generation of misuse or abuse cases as a minimum starting point
- domain-specific security checklists, and an accessible step for beginners
- SecurityRAT, an online tool provided by the Open Web Application Security Project (OWASP) that suggests security requirements depending on the type of application

A more structured approach called Security Quality Requirements Engineering (SQUARE) is provided by the SEI.[2]

The SEI's Mission Thread Workshop can be applied with a security focus by including appropriate subject-matter experts.[3]

[2] See https://resources.sei.cmu.edu/asset_files/FactSheet/2016_010_001_502988.pdf for more information.

[3] See https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=63148 for more information.

## 3.2 Requirements Analysis

There are few requirements analysis tools currently available. Each tool that is available requires training and expertise to apply. Other techniques can be applied, such as assurance cases and subject matter from the SEI's Quality Attribute Workshop. General-purpose tools can be applied by including security-aware subject-matter experts, such as those in mission/business thread workshops.

## 3.3 Architectural Design

More options are available during architectural design. This is an appropriate stage to consider the use of secure libraries and safer programming languages, while other tools require more expertise. Consider the use of the following:

- Assurance cases – These can continue to be developed during design.
- Architectural Tradeoff and Analyses Method® (ATAM®) – This structured approach can be used to analyze, measure, and evaluate how the architecture supports the requirements. This technique requires a trained leader and architecture team and presumes a Quality Attribute Workshop has been performed.
- Software Engineering Risk Management – This analysis is a structured approach with a specific security focus.

## 3.4 Implementation

Implementation includes detailed design, code, and unit test. Many of the technical objectives require direct attention during implementation, including those listed below.

- Provide design and code quality.
- Counter known vulnerabilities.
- Counter unintentional "like" weaknesses.
- Counter intentional "like" weaknesses.
- Counter development-tool-inserted weaknesses.

Other technical objectives that may be addressed are listed below.

- Provide secure delivery.
- Provide anti-tampering.
- Ensure secure configuration.
- Ensure access control.

There are many options for tools and techniques during this phase. The application layer is obviously critical to protect as a last line of defense. However, since seemingly non-critical components can be used for exploits, care should be taken with all software placed on the system.

Studies show that the most cost-effective techniques are checklist-based manual spot checks and formal peer code review inspections. However, to be effective, training is normally necessary.

At a minimum, use the following:

- static code quality and source code weakness

- warning flags on the compiler
- version control
- negative test cases

Moreover, automate as much as possible. Automation not only saves time but assures that the tools and techniques are consistently applied.

Source code quality and weakness analyzers should be integrated into the development, preferably after spot checks and peer inspection. These tools are equally effective in any subsequent development activity. Regression test cases should be automated to prevent known defects from recurring.

Binary/bytecode weakness analyzers do not require special expertise to use and have been found to be very effective when performed as part of the build. However, the source code is necessary to fix, rather than simply identify, residual issues. Because the defect fix effort in binary/bytecode check is somewhat greater than in static code analysis, static code analysis should be run first. Binary analysis, however, can be performed before or after test with minimal change in total rework cost.

If your product contains significant portions of commercial or open source code, strongly consider the inclusion of origin analysis to assure that the most current known weaknesses and vulnerabilities have been addressed.

Unit-test-coverage analysis can be very sophisticated, but line-of-code-coverage analysis is a good start. Some regulations (e.g., DO-178C) require complete modified condition/decision coverage (MC/DC) test.

## 3.5 Integration

Integration is the process in which lower level components are assembled into a system. This process can be repeated, integrating systems into systems of systems. This process may take place with or without the source code. In many contexts, at least the first level of integration is performed by the developers or the developing organization. The key technical objectives are to

- provide design and code quality
- counter weaknesses inserted by development tools

The assembled code also provides opportunity to address the other technical objectives from implementation, such as to

- counter known vulnerabilities
- counter unintentional “like” weaknesses
- counter intentional “like” weaknesses

### 3.5.1 If Source Code Is Available

At a minimum, version control should be used to assure the build is configured as intended. This step can be at least partially verified with origin analysis. It is also a good practice and inexpensive to verify the build with component signatures maintained in the revision control system.

This is the first opportunity to apply binary weakness analyzers to the integrated product.

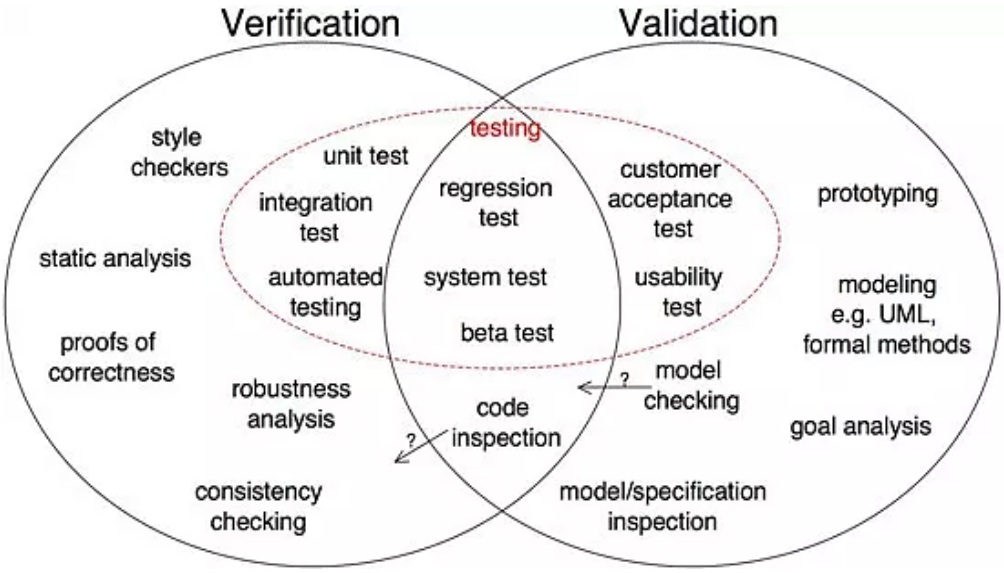
### 3.5.2 If Source Code Is Not Available

Binary weakness analyzers can still be applied during the integration, though the remediation options are limited to identifying weak components.

## 3.6 Verification Process

While the overarching objective of verification is to provide design and code quality, all technical objectives for implementation still apply.

Verification activities assure that the product performs correctly as specified. Although verification is often thought of as testing of the integrated product, the verification process includes a broad range of activities that must be employed in parallel throughout all development processes. There is some overlap of verification and validation (see Section 3.8), with some tools being applicable to both. Figure 9 shows the overlap of categories for many tools and activities.




reprinted from https://www.easterbrook.ca/steve

*Figure 9: Venn Diagram of Verification and Validation Activities*

This guidebook includes many of the verification activities within scope of the development processes because they are typically performed along with the implementation work. Nonetheless, verification can be performed at any time.

The verification of the integrated components includes system test. At a minimum, include negative test cases, and we strongly recommend maintaining an automated regression test suite.

Static weakness analysis of the binary/bytecode should be included if possible. Traditional virus and spyware scanners can also be applied at this point.

Dynamic tests such as fuzz testing and coverage-guided fuzz should be considered. These tools are very effective at exposing buffer issues, and there is some evidence that they are highly efficient at detecting weaknesses.

## 3.7 Transition Process

Transition is the process of deploying the system into its operational environment. The key technical objectives are to

- assure secure delivery
- assure that the system has not been tampered with

Even if developers are not directly involved in the transition, they need to prepare the product for transition.

### 3.7.1 If Developers Perform the Transition

In this scenario, the development should, at a minimum,

- produce a digital signature manifest consistent with the revision control
- verify the delivered software and components manifest matches their digital signature
- verify the permissions from the permissions manifest

Attack modeling of the transition is also recommended. If the rebuild-and-compare technique is used, the transition site will also need access to the revision control and build environment.

### 3.7.2 If Developers Do Not Perform the Transition

In this scenario, the development should, at a minimum,

- make sure the software and components contain a digital signature
- supply a component manifest
- supply a permission manifest

If a rebuild and compare will be used, development must also provide the source code and build environment.

The *SOAR* includes descriptions of the following techniques and countermeasures applicable to the transition process:

- attack modeling
- compare binary/bytecode to application permission manifest
- digital signature verification
- execute and compare with application manifest
- host-based vulnerability scanner
- origin analyzer

## 3.8 Validation Process

Validation involves assessments of how well the software addresses the real-world need when used in the target environment. All the technical objectives from verification remain, with additions.

Validation is often thought of as acceptance testing, but validation activities also include requirements modeling, prototyping, and user evaluation.

Developers must employ validation process activities during the requirements elicitation, requirements analysis, design, and implementation processes.

Validation is sometimes thought of as the final product validation (user acceptance) that normally occurs during or after transition and may not directly involve the developers. This is an important step, but it is only one of the validation activities. Developer validation activities should include the following, at a minimum:

- review of security requirements with stakeholders
- review of the design and implementation of security requirements
- development of negative use cases (e.g., abuse, misuse)
- prototyping
- regular demonstrations of the software to stakeholders prior to release

At a minimum, final validation should include negative test cases. We also recommend attack modeling and the documentation and inspection of the assurance cases.

User test can employ the following:

- fuzz testing
- penetrating testing
- inter-application flow analyzers

## 3.9 Operation Process

A number of tools and techniques are available to support operations. Developers may or may not be involved in the operation. For the purposes of this guidebook, most operational concerns are outside the developer scope. Nonetheless, since developers are involved during maintenance, they need to be aware of the operational environment and application use.

Key objectives include verifying that the system has not been tampered with and monitoring the system for suspicious activity. The specific techniques depend on the deployment platform.

At a minimum,

- provide facilities to verify product integrity (e.g., verifying the digital signature)
- review user reports and operational logs for suspicious activity or changes in product use

## 3.10 Maintenance Process

The maintenance process includes modification of a software product after delivery to correct faults, improve performance or other attributes, or adapt the product to a modified environment. All activities from development apply to maintenance, but special emphasis must be placed on the emerging threats, newly discovered weaknesses, and changes to components such as libraries.

In addition, at a minimum,

- maintain the regression test suite

- regularly evaluate operational logs for new or realized threats
- regularly review and update the attack model
- assure newly discovered weaknesses and vulnerabilities are addressed
- obtain updates to source code and binary/bytecode analysis tools
- evaluate the effect of changes in the operational environment or product use

The technical objective of countering like weaknesses is a moving target. Lists such as the CWE and CVE must be monitored, and static analysis tools must be kept up to date.

In addition, consider using source code knowledge extractors, which are especially helpful if the original developers are no longer available.

## 3.11 Communicating Software Security Assurance

From time to time, and particularly at milestone events, the program manager is required to provide stakeholders with evidence for a justified level of confidence in the software security assurance. Direct demonstrations are inadequate because of the almost limitless ways that attackers might try to exploit weaknesses or vulnerabilities. Since it requires an impractical amount of time and effort to demonstrate that software is adequately secure, developers must be prepared to show their work.

The stakeholders (e.g., the user or project manager) must also see visible evidence that careful attention to security has been built into the workflow. Evidence comes from plans and development data and artifacts. To make the work externally visible, compose a workflow that addresses security assurance, then document the application of tools, techniques, countermeasures, and results. Much of the needed evidence can be automatically gathered by building it into automated parts of the development workflow.

Discuss the reporting and evidence needs with project management. Review the security plan and progress with the project manager. Include security assurance along with progress, cost, and schedule reports and in demonstrations.

More detail on measurement is the subject of the following section. This section suggests some steps developers can take to demonstrate compliance to the project manager.

- Document a security plan that analyzes the threats and countermeasures; for example, provide a presentation to the project manager.
- Keep an inventory of security assurance tools, techniques, and countermeasures and how these are integrated into the development process.
- Keep records of any inspections of documents, designs, code, test cases, or process automation. Capture the product under review, its size, when the event happened, the number of people involved, the duration, any checklists or standards used, and the outcomes of the review. This step is especially important to demonstrate adherence to secure coding standards and design objectives.
- Automate tools into the workflows wherever possible, both to assure tools were used consistently and to automate storage of outputs. Automation includes but is not limited to test cases, builds, revision and configuration control, integrated development environments (IDEs), and so forth.

- Store and maintain the log of tool findings along with their prioritization, remediation, mitigation, or disposition.
- Log reports from vulnerability scans.
- Report results from test coverage reviews or automated analysis.
- Maintain a change log or revision control for test cases.
- Present the process for handling problem reports, including prioritization and disposition. Keep records in an issue-tracking system.
- Summarize the density of findings from reviews, tests, and other tools (e.g., the number of issues divided by product size in function points, pages, or lines of code).

Before beginning work, present the initial security work plan to the project manager and other stakeholders for their information and concurrence.

During work, report changes to or variances from the proposed security assurance work. Report security assurance activities and results on the same footing as time, cost, and schedule. This gives stakeholders confidence that the definition of "done" includes full attention to security assurance.

# 4 Measuring Software Assurance

Measurement is the objective observed value related to an object or event. A metric is any observed or calculated value related to an object, event, or set of objects/events. A useful set of metrics typically contains several actual measurements along with several and possibly many more calculated values based on those measurements [Black 2008].

Metrics contribute to good decision making, but they also present challenges. Without numbers (i.e., measurements), you're only guessing, so use a set of metrics to inform your engineering judgment, not to replace it. The following guidelines may help.

1. It can be difficult to measure consistently (i.e., to take or make a measurement), so do not overreact to noise in the metrics.
2. Many useful metrics do not directly measure the intended target. Instead they measure a "proxy" that stands in for the target, so you must make some assumptions.
3. Do not rely on a single metric for truth.
4. Many useful metrics are derived from other metrics or multiple measurements. If any of the component values use different assumptions, the metric may be of limited usefulness or simply invalid. Check your assumptions.
5. Reliance on any given measurement or metric can lead to improved results with respect to that number without actually achieving the real objective.

Metrics that might be used for software assurance include those for measuring operational risk, actual attacks, potential exposure, and cost of mitigation and remediation. For the purposes of this guidebook, we focus on metrics that may be useful in selecting and applying tools or techniques during development.

It is useful to think of measures along certain dimensions. Measures can be explicit or derived. Explicit measures are taken directly, while derived measures are computed from other explicit or derived measures. Examples of explicit measures include a count of vulnerabilities discovered, total lines of code, or number of input sources. An example of a derived measure is the number of vulnerabilities per file or vulnerabilities per thousand lines of code [Humphrey 1995].

Measures can be predictive or explanatory. Predictive measures can be obtained or generated in advance, while explanatory measures are produced after the fact. While explanatory measures describe what happened, predictive measures describe what will (or is likely to) happen.

Measures can be absolute or relative. Absolute measures are typically invariant to the addition of new items. The count of vulnerabilities in a file, for example, is an absolute measure independent of the defect count of other files. A relative measure places the absolute measure into the context of other measures. For example, a benchmark from similar programs might be used to characterize the defect density as average, high, or low.

Measures can be objective or subjective. Objective measures count things, and subjective measures involve human judgment. A CWE vulnerability density of one per million lines of code

is an objective measure. A survey of users rating a product's security (e.g., using a Likert scale) is subjective.

Finally, measures are often made using proxies. A proxy correlates with the subject of measurement and therefore stands in for another measure. The correlation may be strong or weak. The relationship may be directly causal, for example, a coding defect leading to a weakness. The relationship may be part of a causal chain: increased operational load may lead to a fault because of an increase in the number of paths executed, some of which are faulty. There may be some common cause; stricter compiler warnings find both low-severity syntax errors and common weaknesses. All types of measures have uses. However, misusing measures can lead to dysfunctional behavior [Austin 1996].

## 4.1 Software Security Measurement

Unfortunately, there is not yet much empirically grounded guidance on which security metrics are useful. To make the most of what is available, apply a form of the Goal, Question, Metric approach [Basili 1992, Park 1996, Goethert 2004], summarized as follows:

1. Establish the goal or objective.
2. Determine what questions, if answered, would help to achieve that goal.
3. Find quantitative measurements that help to answer the questions.
4. Validate the metrics.

For example, your objective may be to convince a customer that you are taking reasonable steps to avoid field incidents. One (among many) of the questions you can ask is "What available techniques am I using or not using?"

One commonly used set of metrics involves a count of tools and techniques in different classes. These are often used in conjunction with the BSIMM. These counts can be compared to technical objectives and compared with peer groups in the industry. Of course, counts alone do not assure that the practices chosen are effective for the system, nor do they assure that the practices are applied effectively. Nonetheless, a tool type analysis provides a list of potentially effective tool options and is a starting point for additional analysis.

Measures of effectiveness can also begin with counts. If we frame our objective as "use the tools that will be most effective," the questions we ask will be framed around how to measure that effectiveness. We know that certain tools are more or less effective against specific weaknesses. For example, we can count CWE finds and unique types found during static analysis. If we want to compare different components or products, we might use size to develop a weakness removal density. To make measurement outcomes more relevant, we could count discovered vulnerabilities during operations, perhaps as a weakness (or vulnerability) discovered during a time period.

A problem with vulnerabilities discovery during operations is that they tend to be rare. When analyzing rare events, it is often hard to identify real trends or take decisive action. Other types of defects may predict vulnerabilities and occur 20 to 100 times as often [Woody 2014]. Defect data from inspections, unit test, and system test have been found to correlate with vulnerabilities in at

least some systems. Moreover, these measures are predictive, allowing time to take corrective actions if high levels are discovered.

We can count all the tool findings and how they were disposed. Measures of weakness findings from static analysis present additional opportunities. Unlike test or inspection, static analysis findings are precisely reproducible for the same inputs. Nonetheless, static analysis will also consistently not find other issues. Experience with compilers suggests that the density of findings (per line of code, function point, or file) for static analysis will have at least modest correlation with issues that the tool did find. High levels of findings with any imperfect tool strongly suggest applying additional tools and techniques.

Our objective may be to use our limited resources efficiently to remove as many weaknesses as practicable. What removes the most defects in the shortest time or least effort? To answer this question, measure the effort required to apply tools and how much time was spent in addressing issues. Some tools are more or less efficient. For example, static source code analysis points directly to a problem, while negative testing will show the presence of a deficiency, but not the source. However, static source code analysis will also find a variety of issues, many of which may not be severe. It usually pays to remove as many issues as possible before test, but to be sure, measure the effort, defects, and vulnerabilities separately and compare.

## 4.2 Short List of Basic Security Metrics

Take and use measures to help make decisions. This guidebook groups some common metrics by use, though metrics often address more than one use or use category. The primary measures address the following questions:

- How big is the product or change to the product?
- What was done?
- When did it start and end? (effort and schedule duration)
- What defects or vulnerabilities were injected or removed?

### 4.2.1 Product Metrics

Product size is often helpful to determine the amount of effort required to build or maintain a software system. More detailed knowledge helps when selecting needed personnel skills and developing infrastructure. Product size is usually obtained through the build system and configuration management. Common size metrics include

- product size (usually in lines of code or function points)
- changed size during a release, usually total lines added, modified, and deleted
- number of components
- number of components changed for a release
- number of components by category of origin (COTS, GOTS, OSS, and so forth)
- defects reported
- vulnerabilities discovered
- density of weaknesses (or defects)

### 4.2.2 Responsiveness

Responsiveness measures the amount of time required to address issues once they become known. The responsiveness metric helps to assess the amount of time during which the product and users remain exposed to a vulnerability. Responsiveness helps to assess risk and measure maintenance effectiveness. Responsiveness metrics can help developers make trade-offs when deploying resources or assessing process changes. Responsiveness metrics include

- time between patch of COTS/GOTS component for vulnerability upstream and deployment of patched software
- time between discovery of vulnerability and deployment of patched software

### 4.2.3 Process Effort Metrics

Process effort establishes historical baselines to be used in planning allocation of resources. Process metrics can also be compared to benchmarks to determine if the types of work are consistent with good practices. For example, was something missed, or perhaps performed superficially? Was the effort applied using particular techniques excessive?

A typical use is to not only estimate the cost and schedule required for a patch but also to determine the allocation of any special skills or equipment, such as vulnerability analysis, testing, and so forth. Typical metrics include

- techniques used during each development activity
- effort fixing defects, weaknesses, and vulnerabilities
- effort applying security-related tools and techniques
- test effort and duration
- rate of weakness (or defect) removal

### 4.2.4 Effectiveness

Effectiveness metrics help to determine if techniques are achieving the desired results. Metrics include

- fraction of weaknesses removed by a technique
- ratio of defects or vulnerabilities found in production compared to those found in test

### 4.2.5 Test Metrics

Test metrics help when assessing if the test process is adequate. Common metrics include

- line-of-code coverage (usually expressed as a percentage of lines executed)
- path coverage, which includes the number of program paths covered and the total number of execution paths available [RTCA 2012]
- number of test cases and number of test cases for each requirement
- test failures per build
- counts of defects, weaknesses, or vulnerabilities. These usually have additional characteristics such as CWE reference, orthogonal defect classification type [Chillarege 1992], and activity during which they were discovered.
- counts and density (usually by lines of code) of defects by component

- potential severity of the weakness (e.g., Common Vulnerability Scoring System, or CVSS, score)

Additional measures or estimates will be required for risk analysis, but this requires more discussion that is beyond the scope of this guidebook.

## 4.3 Measurement Resources

*Table 3: Resource List for Measurement in Software Assurance*

| Resource | Description |
|---|---|
| *Cyber Security Metrics and Measures* [Black 2008] | Handbook on cybersecurity metrics published by NIST |
| *Goal Question Metric (GQM) Paradigm* [Basili 1992] | A premier resource for measurement in software engineering |
| *Goal-Driven Software Measurement — A Guidebook* [Park 1996] | A guidebook to help identify, select, define, and implement software measures to support business goals |
| *Personal Software Process* [Humphrey 1995] | A book by Watts Humphrey that provides a highly useful implementation for measurement in software |
| *The Economics of Software Quality* [Jones 2011] | A book by Capers Jones that addresses macroscopic issues |
| *Software Quality Metrics Overview* [Kan 2002] | A short introduction to measurement theory and application |
| *Integrated Measurement and Analysis Framework for Software Security* [Alberts 2010] | An SEI report that provides security metric resources |

# 5 Guide to the State-of-the-Art Report (*SOAR*)

*State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation*, published by the Institute for Defense Analyses, assists DoD program managers and their staff members in making effective software assurance and software supply-chain risk management decisions, particularly when they are developing their Program Protection Plan (PPP). The *SOAR* contains a comprehensive list of tool types, characteristics, known effectiveness, and other information. Because the volume of information in the publication is so large and its scope so broad, this section provides summaries of the chapters and guidance for the tool selection process it describes.

## 5.1 Chapter Summaries

### Chapter 2: Background

This chapter of the *SOAR* defines key terms that are used throughout the text. Especially important in our context are terms involving evaluation artifacts and the findings from running software assurance tools on those artifacts. The language of the evaluation results makes clear that the tools are imperfect and will identify false positives (i.e., reported findings that are not weaknesses) and false negatives (i.e., not all weaknesses are identified in report findings). Some key terms are defined below.

*Targets of evaluation (TOE)* are the artifacts that can be examined during the development process. Because tools and techniques are usually specific to artifacts, the available artifacts are essential to composing software assurance workflows that include security considerations. TOE can include design, source code, object libraries, test cases, executable programs, and environments.

*Source code* is the set of computer instructions in a human-readable computer language that is written and maintained by software developers. Source code is often translated into a *bytecode* or *binary* (using a program or device called a compiler). Binary is explicit computer instructions that are executed directly on a specific computing platform. Bytecode (for example, from Java or .Net) is an intermediate representation that is input to another program for execution. The distinctions are important, in part because commercial products included in the development, such as libraries and frameworks, may not include source code.

*SOAR* includes additional terms used by security tools. A *site* is where a weakness might occur. A *finding* is a definitive report about the site. A complete tool produces a finding for every site. A *sound tool* is one for which *every finding is correct*. Part of the challenge of workflow composition is that (1) not all sites necessarily contain findings and (2) not all findings are necessarily correct. Moreover, different tools may contain overlap in the sites and findings.

A *true positive* is a finding that represents a weakness that must be corrected. A *false positive* is a finding that is not a true weakness or a weakness that cannot be exploited. A *true negative* is where a site contains no finding of a weakness. A *false negative* is a site for which a weakness is

present, but it escapes the finding. Understanding the rates and remediation cost of true/false positives/negatives will become important when composing processes.

Other key terms from the *SOAR* appear in the glossary in the back of this guidebook.

## Chapter 3: Overall Process for Selecting and Reporting Results

### Approach

The *SOAR* summarizes an overarching approach to use when selecting tools and reporting the results from their use. In short,

1. Identify the technical objectives (*SOAR*, Chapter 4).
2. Select tool and technique types (*SOAR*, Chapter 5).
3. Select specific tools (guidance in *SOAR*, Chapter 5, and the fact sheets).
4. Summarize the selection (for example, in a PPP).
5. Apply analysis.

### Using the Matrix

The *SOAR* provides a matrix in an appendix with a rich but dense description of the techniques appropriate to various technical objectives. Dimensions of data presented include

- technical objectives
- lower level objectives
- types of tools and techniques appropriate to the technical objective
- cost to implement
- cost effectiveness
- best applicability

In principle, this matrix provides sufficient information for an expert to select tools. In practice, experience and additional guidance are likely to be necessary to use the matrix effectively. The information is so dense and the matrix so large that it is easy to become overwhelmed. Printed out in readable text, the matrix is the size of a large wall poster. The SEI is developing additional job aids, such as the Project Context Questionnaire (see Appendix F), to help make these decisions.

## Chapter 4: Technical Objectives

This chapter of the *SOAR* describes the role of technical objectives in tool selection. Because tools can be used on different artifacts and can find different kinds of vulnerabilities, to select specific tools, the intended technical objective should be matched to the tool capability. The first step is to operationalize “security” by selecting the technical objectives.

The overarching objective of security can be decomposed into more specific objectives of confidentiality, integrity, availability, authentication, and non-repudiation. Connecting these objectives to actions, however, is more difficult because vulnerabilities can affect multiple categories. A bottom-up approach (see Section 6 in this guidebook) is thus necessary to avoid weaknesses and vulnerabilities that can affect security.

The *SOAR* goes on to note that for more specific categories, such as the CWE, the structure does support the selection of specific tools for process composition. Objectives depend on the software lifecycle stage and activities performed. The primary stages are listed below.

### Development Stage

- Provide design and code quality. It recognizes that quality problems can manifest as weaknesses and vulnerabilities. While no specific guidance is provided regarding cost and benefit, expert opinion has converged on the belief that general quality is a necessary condition for security.
- Counter known vulnerabilities. This essentially advises us to examine for known vulnerabilities, including but not limited to those in the CVE. This also implies being aware of existing issues in components and packages used in the development.
- Counter unintentional-like weaknesses. This is similar to addressing code but focuses specifically on known weaknesses such those from the CWE top 25 list.
- Ensure transparency.

### Acquisition Stage

- Ensure transparency.

### Pre-Development/Design Stage

- Provide design and code quality.
- Ensure authentication and access control.
- Provide anti-tamper and ensure transparency.

### Operational Stage

- Counter intentional-like malicious logic.
- Provide secure delivery.
- Provide secure configuration.

## Chapter 5: Types of Tools and Techniques

The *SOAR* categorizes tools into three analysis classes: static, dynamic, and hybrid. This chapter explains the categories of tools and the technical objectives to which they can be applied.

**Static analysis** examines the system without executing it and can be applied to design representations, source code, binaries, and bytecode. Tools include attack modeling, source code analyzers, obfuscated code detection, bytecode or binary disassembly, human review/inspection, origin analysis, digital signature verification, configuration checking, permission manifest analysis, development/sustainment version control, deliberate obfuscation, rebuild and compare, and formal methods.

**Dynamic analysis** examines the system execution, giving it specific inputs and examining results and/or outputs. Tools and techniques include network scanners, network sniffers, network vulnerability scanners, host-based vulnerability scanners, fuzz testers, framework-based fuzzers, negative testing, digital forensics, intrusion detection systems and intrusion prevention systems,

automated monitored execution, forced-path execution, firewalls, man-in-the-middle attack tools, debuggers, and fault injection.

**Hybrid analysis** applies to the tight integration of static and dynamic analysis approaches.

No one type of tool or technique can address all possible technical objectives.

### Chapter 6: Software Component Context

In this chapter, the *SOAR* indicates that the PPP should list the context factors that distinguish the components or the use to which they are put. These contexts help establish the risk to the mission and opportunities for security analysis.

Use factors include

- mission criticality
- information availability
- critical program information

Software product factors include

- amount of custom development (e.g., COTS, GOTS, OTS)
- specific technologies
- programming languages
- supply chain exposure
- operational usage

Development environment factors include

- compiler
- runtime libraries
- automated test system
- configuration management system
- database

### Chapter 7: The Program Protection Plan Roll-up

This chapter of the *SOAR* explains how to document that the security assurance plan meets the mission and technical objectives. The purpose of the PPP is to help programs ensure that they adequately protect their technology, components, and information. The process of preparing a PPP is intended to help program offices consciously think through what needs to be protected and develop a plan to provide that protection.

### Chapter 8: Application

The application section recommends ways to apply the processes described in the *SOAR*. The topics include selecting technical objectives and selecting combinations of tools and techniques. The top 10 technical objectives are listed and described in the chapter.

When selecting technical objectives, the *SOAR* suggests you begin using these steps:

1. Decompose the system into distinct parts to identify those that are sufficiently critical to merit further analysis.
2. Identify the types of vulnerabilities that apply to the component.
3. Determine and prioritize technical objectives.

The *SOAR* notes that when selecting tools, many address multiple technical objectives. A minimal subset of practices is then listed, including the following:

1. Use tools that are both inexpensive and effective, including attack modeling, warning flags, traditional virus scanners, and hardening tools and scripts.
2. Consider safer languages for greenfield development.
3. Implement source code quality and weakness analyzers, which can be effective and easy to use.
4. Consider origin analyzers for systems built with third-party components.
5. Apply manual spot checks.
6. Consider web-based scanners for applications and servers.
7. Use fuzz testing variants for systems not covered by web application scanners.
8. Include negative test cases in the test suite.
9. Use test coverage analyzers to assure adequate test coverage.
10. Use digital signature verification to assure that the deployed versions have not been tampered with.

### Chapter 9: Vignettes

This chapter of the *SOAR* offers several examples demonstrating how to identify the software component context, select technical objectives based on that context, and select tool and technique types to address the technical objectives. It also captures some common-sense guidance for systems composed of OTS proprietary, open source software, and custom critical components.

### Chapter 10: Gaps

Key gaps identified in this chapter include

- finding unknown malicious code (i.e., once malicious code is known, it can be recognized)
- integrating different tools. While in principle tools examine different aspects of a system and should provide better security when used in combination, they have different reporting mechanisms and the benefits of combinations are unknown.
- use of quantitative data. Quantitative data is very limited, and there is little ground truth for making decisions.
- the relationship between quality and security.
- security measures. Although measures of security exist, they may be poorly defined or proxies of unknown correlation with the actual target of measure.

### *SOAR* Appendices

The bulk of the *SOAR* is included in its appendices, and a useful companion spreadsheet listing tools types, technical objectives, and other data is also available. *SOAR* appendices include the following:

- Resources Used
- Key Topics Raised in Interviews
- Fact Sheets
- Detailed Compositional Views
- Software State-of-the-Art Matrix
- Mobile Environment
- Additions to the 2012 *SOAR*
- Acronyms
- Bibliography

## 5.2 The *SOAR* Tool Selection Process: A Top-Down Approach

The following sections summarize material from the *SOAR* about selecting tools. The approach described in the *SOAR* is "top-down," starting from project characteristics. This summary is intended to provide an overview of the recommended approach for tool selection, along with some additional considerations. Consult the *SOAR* for more detail if you determine this is a useful approach to choosing tools for your project.

### 5.2.1 Overview

The *SOAR* recommends the following steps for tool selection:

1. Identify the software components in a TOE, and determine each software component's context of use (as described in *SOAR*, Section 6.A).
2. Determine the following for each software component context of use:
    a. Identify the technical objectives based on context (*SOAR*, Chapter 4).
    b. Select the tool/technique types (*SOAR*, Chapter 5) needed to address the technical objectives, using the matrix (discussed in *SOAR*, Section 3.B, and presented in *SOAR*, Appendix E).
    c. Select specific tools (see guidance in *SOAR*, Chapter 5, and the fact sheets in the appendices).
    d. Summarize selection, which may be part of a larger report. In the DoD, this would be part of the PPP (*SOAR*, Chapter 7).
    e. Apply the analysis tools, use their results, and report appropriately. Here the selected tools and techniques are applied, including the selection, modification, or risk mitigation of software based on tools (*SOAR*, Chapter 7).

### 5.2.2 How to Implement the *SOAR* Process

Implementing the selections suggested by the *SOAR* requires a substantial effort. We recommend the following structured approach if you choose to do so:

1. Baseline the development and use context, and record the results. This step identifies
   a. critical assets (i.e., potential TOE)
   b. key cost and security drivers
   c. overarching technical objectives
   d. recorded history of the key context factors
2. Baseline development costs, quality, and initial vulnerability risks.
3. Establish the higher level objectives based on regulation, business requirements, or both. These can include goals associated with a vulnerability target, compliance mandates, requirements for documenting assurance, or cost and schedule constraints. In practice, residual vulnerabilities may be accepted within the risk management framework.
4. Use the TOE, lifecycle stage, and overarching technical objectives to down-select a list of potential tools or technique types that could apply. Because tools are typically applied to specific artifacts or development activities, the available tool type list can be narrowed.
5. Use the context factors, specific build process, staff expertise, development tools, cost, and expected benefits to select specific tools for implementation. Security tools are often specific to development tools, programming languages, and so forth. Specific tools can be selected as candidates for use based on these local context factors.
6. Consider implementation costs, use costs, expected benefits, and staff experience to select specific tools.
7. Measure the results of using the tools on acquisition cost, schedule, development cost, and maintenance costs.

A key consideration for selecting software assurance tools is the system lifecycle stage. The term "lifecycle" has multiple distinct meanings and representations. The DoD acquisition lifecycle, with callouts for software assurance, is represented in Figure 10. In practice a product in production and deployment is adapted, improved, and corrected, such that individual components may re-enter the development lifecycle stages and lifecycle processes iteratively.

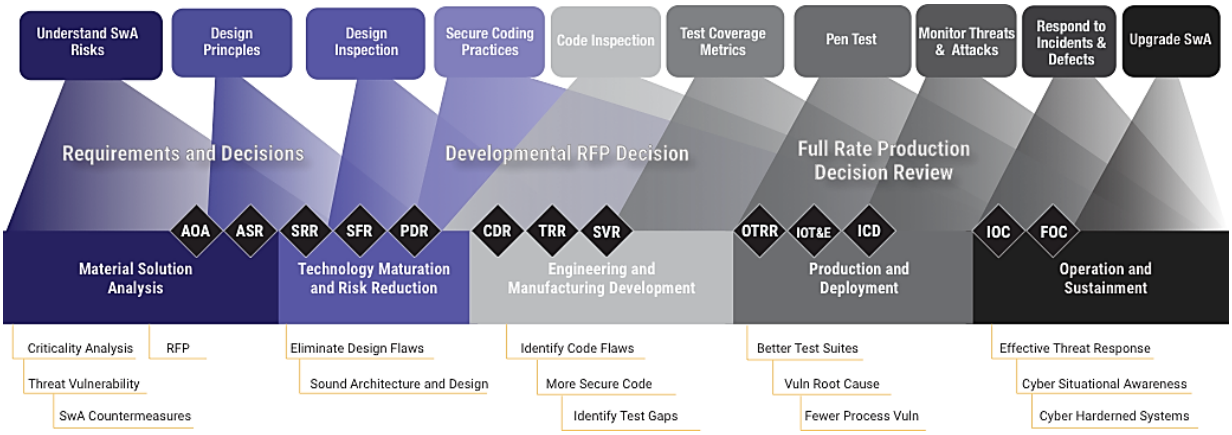



*Figure 10: Software Assurance for DoD Systems*

### 5.2.3 Steps for Selecting Tools

The steps outlined below provide more detailed suggestions for selecting tools and techniques consistent with the general *SOAR* approach. Note that the key tool provided by *SOAR* is the matrix in Appendix E.

#### Step 1: Complete the Project Context Questionnaire

Appendix F includes a form to record the relevant factors to guide the decision process. Complete this form and record the results. Refer to the questionnaire in later steps.

#### Step 2: Baseline the Initial Performance

The baseline provides a point of comparison to evaluate changes introduced by new tools and techniques. While the project context is primarily qualitative, the baseline should include quantitative data on the development (or acquisition) process, activity costs, schedule performance, production rates, and defects and vulnerabilities found in development, test, and production.

In developing the baseline, refer to the project questionnaire to identify potential sources of data, including requirements management tools, revision control, defect tracking, automated builds, and automated tests. To evaluate the effectiveness and efficiency of the tools in your environment, pay particular attention to effort in test, issues found in test and deployment, and effort mitigating issues found in test or deployment.

Finally, examine issues from test and deployment to identify problems with potential security implications. For those problem findings (weakness vulnerabilities) that are most frequent or individually time consuming to fix, use judgment to determine if the issues could have been avoided or found earlier. If so, identify earlier stages at which tools or techniques could be applied. Typical examples include (but are not limited to) the following:

1. Consider the use of revised threat and attack surface modeling to identify items to secure or environmental conditions.
2. Architectural design often involves interactions with the environment, use of frameworks or packages, process or user privilege, and tradeoffs such as performance, usability, and security.
3. Detailed design and coding typically involve implementation-level defects such as unprotected memory, buffer overflows, memory management, and so forth.
4. Test issues often include failure to test for what the software should not do (i.e., negative testing).
5. Third-party packages may introduce vulnerabilities, especially if they are not current.

#### Step 3: Establish Overall Goals

From this baseline, consider the costs and potential benefits of reduced vulnerabilities at each stage of the process. Develop initial goals for

- total development cost
- total cost of ownership, including deployment and maintenance

- schedule or delivery objectives
- targets for vulnerabilities discovered during development and deployment
- satisfying regulatory or legal requirements

### Step 4: Identify Artifacts and Potential TOE

TOE include all aspects of the software components to be built or acquired, including

- architecture documents
- design documents
- user documents
- source code (by language and age)
- binary or bytecode libraries
- executable bytecode
- executable binary
- test cases
- deployment scripts

The technical objectives are categorized in levels; the full table from the *SOAR* is in this guidebook (see Appendix D). The "Applying" section includes a concise description of the objectives. Record which artifacts are under development or available at each stage of the lifecycle. Specific artifacts can be evaluated with different tools; for example, a number of tools can evaluate source code. Many techniques require multiple artifacts (e.g., the source code and the compiled code).

### Step 5: For Each TOE, Identify Technical Objectives and Preliminary Tools/Technique List

Using the matrix in the *SOAR* appendix, or the summary in this guidebook's Appendix E, match the available artifacts and TOE with desirable technical objectives. Refer to the potential TOE and overall objectives to narrow the choices.

Proceed to more detailed technical objectives based on project-specific context constraints (see Appendix D). For example, some tools require both source code and the built object code. The detailed technical objectives depend on some combination of the product development stage, artifacts available, and risks in use.

The result of this step is a list of targets, technical objectives, and relevant tools that can be applied.

### Step 6: Down-Select the Tool/Technique List

Not all tools and techniques are practicable or cost effective. The *SOAR* includes information on subject-matter expertise and cost to implement. The guidance on cost effectiveness is limited. Use the *SOAR* to make qualitative judgments, gather additional information on implementation costs, and estimate expected benefits for individual tools.

### Step 7: Select Initial Tools and Techniques

We suggest limiting the introduction of new tools at first to accomplish the following goals:

- Limit tool interaction to simplify evaluation of cost effectiveness.
- Ensure the tool has been implemented properly.
- Determine if the tool is effective in use.

It is difficult to fully understand how the tools will work in combination. A useful combination would be a tool used in development (e.g., a static code quality checker) and a tool that evaluates the deployed product (e.g., penetration test or fuzz test). An example of tools that largely do not interact are a static analysis code quality checker and a tool to scan binary libraries for known vulnerabilities.

Prioritize techniques that identify security issues as early in the process as possible. For example, consider attack modeling in requirements rather than relying on adding a negative test after the fact. Likewise, static code analysis can be applied as an evaluation at acquisition, but it is likely to be more cost effective if applied by developers.

### Step 8: Integrate the Tools into the Environment and Complete the Baseline

As the tools are introduced into the environment, examine existing artifacts from prior development. Note the size of the artifacts, volume of reported issues, true and false positives, effort, and time to apply. This provides an objective point of comparison for future work.

### Step 9: Measure Results for New Development (or Acquisition) and Deployment

This step validates that the tools are being used effectively and satisfy program management's cost–benefit analysis (from the Risk Management Framework). When measuring results, consider the following:

- delivery and deployment schedules
- size of the deliveries
- effort and cost required to implement the tools
- vulnerabilities discovered during development
- vulnerabilities discovered during pre-deployment test
- vulnerabilities reported during deployment
- costs in effort and schedule incurred in test
- cost in effort and schedule for deployment and deployment patches

# 6 Building a Secure Development Process: A Bottom-Up Approach

While the *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* presents a top-down approach to tool selection, this guidebook approaches it from the bottom up, considering what activities and tools are normally appropriate at the different stages of development or product lifecycle. The primary reasons to begin bottom-up are to simplify the initial decision process and to directly engage the people performing the work. This approach allows those performing the work to implement the methods that offer low cost and high effectiveness, while the global issues of coverage and cost can be addressed by program management.

The details of the *SOAR* decision process may seem overwhelming. To simplify them, this section summarizes some specific tool considerations based on the product software development cycle and maturity lifecycle stage in which activities are being performed.

This guidebook does not attempt to document a complete or optimized decision tree for selecting tools because of the following:

- The contextual factors lead to an explosion of decision branches and paths.
- The problem is not easily factored for a decision tree because tools and techniques often apply across technical objective and lifecycle boundaries.
- Reliable data on the cost effectiveness of the tools as applied to each technical objective is not yet available.

## 6.1 Contextual Factors

The context of the product, development, and use guides tool selection and prioritization. The project development context influences and constrains the tools and countermeasures that can be applied and affects their effectiveness, direct cost of use, and indirect cost of use.

The first key factors to consider are the availability of the source code and whether the product is developed largely in-house or includes third-party components. Beyond these, there are many contextual factors that describe the development environment, product maturity, application domain, and operational environment. Because the factors vary considerably, there is no single approach or set of tools and techniques in widespread use.

We therefore recommend listing the relevant project characteristics before tool selection. The Project Context Questionnaire in Appendix F is a form to record these characteristics. In brief, the questionnaire collects information about the following:

- software product lifecycle stage (e.g., amount of new development, enhancement)
- product delivery strategy or approach
- relationship with the customer or user of the software
- deployment platform and environment
- business and application domains

- access or storage of sensitive data
- development activities
- software development methods
- software development tools, including (but not limited to)
  - compilers
  - revision control
  - requirements management
  - intrusion detection environment (IDE)
- product size
- programming languages for new development
- programming languages for included open source software
- binary or bytecode libraries used
- project goals for cost, schedule, and quality
- development staff size and experience

This questionnaire was originally designed to aid in the proper use of data collected for software performance and includes context factors considered to be important for valid scientific and engineering decision making [Petersen 2009]. A completed Project Context Questionnaire should help make the selection process transparent, reproducible, and rigorous.

## 6.2 General Recommendations

This section provides some recommendations to support overall cost and effectiveness. These recommendations do not focus on specific technical objectives or specific tool classes. Instead, a recurring theme is to act at each phase in the product lifecycle and software development cycle.

### Take steps to secure the application layer.

Figure 11 illustrates a useful conceptual view of the layers of security during deployment. The captions suggest that while a large portion of the vulnerabilities lies in the core application layer [Clark 2015], most security effort and expense are directed outside [Feiman 2014]. While we cannot verify the specific numbers, we firmly believe that applications can be more secure at modest expense.

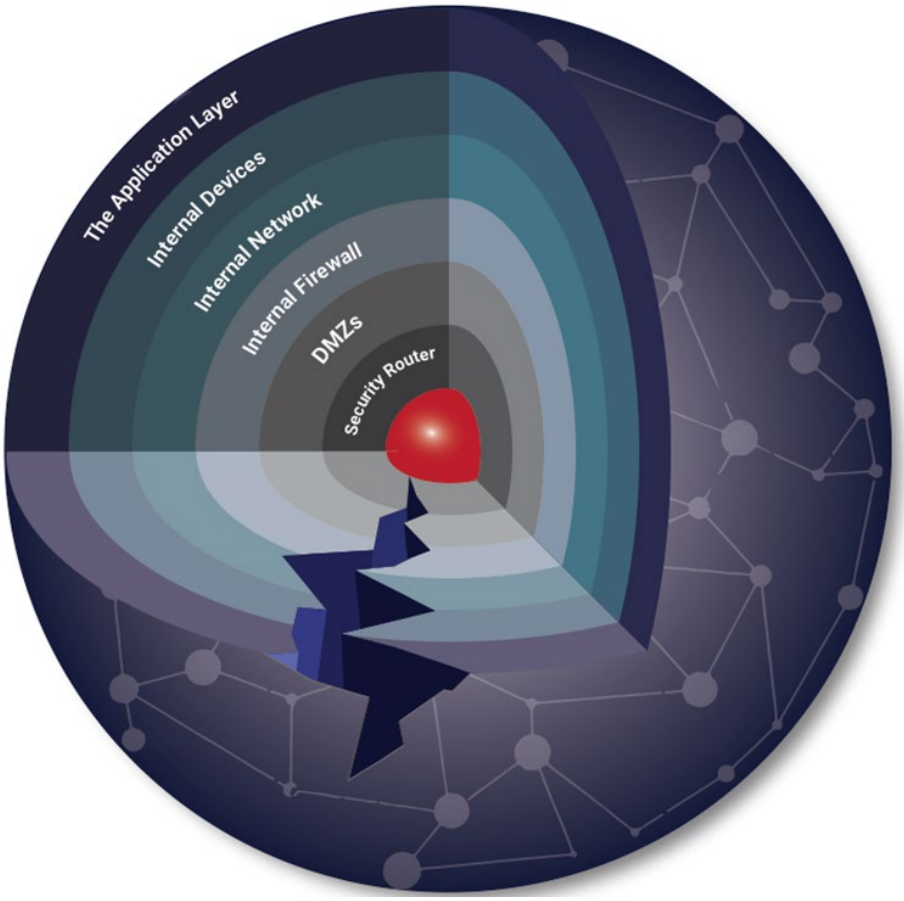



*Figure 11: Layers of Security*

### For applications, focus on quality first.

The quality of the code and design is the number one technical objective and is necessary to ensure the achievement of the other technical objectives.

Research demonstrates the following:

- Poor development quality is a root cause of vulnerabilities. Quality determinations have some predictive capability (see Section 2.3).
- Poor quality can increase, rather than reduce, overall development costs and can substantially increase total lifetime costs and the total cost of ownership (TOC) [Jones 2009, Nichols 2012].

This does not mean that other security activities are unimportant. Instead, we emphasize that software defects are not only a source of vulnerabilities, but will also undermine other efforts in requirements, design, test, and deployment. Defective software will not be secure.

Studies suggest between 1% and 5% of software defects are also security vulnerabilities [Woody 2014]. This is comparable to the fraction of defects that are "Level 1," meaning they can stop a system [Jones 2011]. Because overall defects are far more frequent than vulnerabilities, defect density may provide a more reliable estimate of remaining vulnerabilities than the measured vulnerability density.

This connection among development defects, weaknesses, and vulnerabilities has implications for both developers and acquirers. For all developers and development organizations, this suggests that quality assurance includes managing quality activities and controlling product quality throughout development. Moreover, quality activities must be measured and managed. Acquisition and contracting enhance the need for explicit metrics. Do not rely exclusively on test and tools. Where tools are less than 100% assured of finding a defect, it is prudent to take other measures first. While tools can be very effective for detecting some classes of vulnerability such as known vulnerabilities from the CVE, they are seldom 100% effective. This has been noted anecdotally and in benchmarking of weakness analyzers.

Because test tends to identify the presence of a defect but not the root cause, finding and removing defects in test can be substantially more expensive than other methods, such as inspection and static analysis. Test is therefore an inefficient means for removing large volumes of defects.

Conventional (mostly positive) test is typically only 40–50% effective [Jones 2011], and security vulnerabilities can be even more difficult to identify. More distressingly, 15% of defect fixes introduce new defects, of which only about half will be detected. The fact that defects often mask other defects by altering execution behavior compounds the problem. Finding substantial numbers of defects and vulnerabilities during test should be a red flag that additional vulnerabilities remain undetected.

These concerns imply that projects should consider vulnerability prevention and removal during all lifecycle activities. These steps include reviews and inspections of all project artifacts, including but not limited to requirements, design, code, test cases, and deployment processes and scripts.

### Consider the product lifecycle stage, development activities, and deployment separately and explicitly.

For clarity, we distinguish between software "development cycles" and the product lifecycle. The *SOAR* uses the term so*ftware development lifecycle* and includes development (requirements, design, implementation, and test), deployment, operations, sustainment, and disposal. This mirrors the DoD lifecycle as described in DoD Instruction 5000.02 [USD(AT&L) 2017]. When we use the term *software development cycle*, we refer to the activities (usually involving at a minimum design, code, review, and testing) that transform an increment from a stated need through software working software product. The activities can include multiple tests performed by different teams, unit test, integration test, system test, acceptance test, and operations test. Our use of product lifecycle refers to the age and maturity of the software. The Rational Unified Process (RUP) describes lifecycle stages as inception, elaboration, transition, and retirement. To this we would add a sustainment phase that might include ongoing enhancement, migration, and bug fixing.

The product lifecycle stage affects the relative emphasis on technical objectives and development activities. Moreover, the lifecycle stage often constrains the options available because development cycle stages and artifacts determine which tools and techniques can be applied. The lifecycle stage therefore affects the cost and effectiveness of security technique and tool use.

Although development cycle factors appear in the *SOAR*, we recommend thinking about them explicitly before considering the technical objectives or tool-selection process. For example, legacy code may preclude selection of a safer language, but static analyzers can be applied. However, it is most sensible for static analyzers to be used by those working most closely with the code during construction rather than later.

## 6.3 The Selection Process

Prior to the selection of technical objectives, tools, and countermeasures, we recommend completing Steps 1, 2, and 3 from *SOAR*, described in Section 5.

To help select and prioritize artifacts, decompose the delivered products into distinct components that fit into application and domain categories of the profile survey. A criticality analysis, [Paulsen 2018] can help identify and understand the risk and consequences of failure in subsystems, components, or subcomponents in operation. This will include third-party source code, binary or bytecode libraries, embedded systems, web applications or servers, database applications or servers, and so forth.

Next, complete the project profile to collect relevant contextual information to guide further steps. If you identified distinct components, separate and collect the component-specific information. At this point, and before proceeding to the requirements, some threat modeling should be performed. Note that threat and attack modeling must be integrated into the requirements and design processes. Initial threat modeling must consider risks to assets, potential attackers, attributes of the system, and known or likely components.

Threat modeling will continue throughout the product lifecycle. Because threat modeling techniques have a wide variety need for training, difficulty of use, automatability, tool support, and effectiveness for purpose, consider approaches early. Multiple tools and approaches may be chosen. A useful analysis of the strengths and weaknesses of various threat modeling tools for cyber physical systems can be found in the 2018 report by Shevchenko [Shevchenko 2018a].

At this point you can list the most likely sources of vulnerability for each component in each major stage of the development cycle. Remember that vulnerabilities may be injected accidently or intentionally. Vulnerabilities may be injected during development or by tampering with the product, allowed through the failure to specify and design appropriate security attributes into the product, inherited from OTS, or introduced by using OTS in unexpected ways.

We recommend that teams responsible for a development stage examine each of the technical objectives including requirements, architectural design, component-level design, construction (detailed design, coding, and unit test, including bug fixes) integration, system or acceptance test, and deployment, and use them to determine appropriate tool and countermeasures.

Done separately, this will produce a list of tools that will have redundancy and is likely to be globally suboptimal. Nonetheless, the list is likely to be comprehensive, and a number of tools should be somewhat effective at minimal net cost. Global optimization will have its own challenges and should be coordinated at the program level.

### 6.3.1 Select Development-Stage-Specific Tools

This section lists some tools that are likely to be cost effective during different development phases. Clearly security and policy requirements should be considered, and human review is a minimal technique to apply. Many of the tools support specific technical objectives, which are listed at the end of Appendix D of this guidebook.

#### 6.3.1.1 Requirements Considerations

The requirements process includes eliciting, analyzing, and ranking the requirements associated with the mission. Several techniques can be employed during requirements to enhance security, including the Mission Thread Workshop (MTW), the Software Engineering Risk Analysis (SERA) framework, threat modeling, and attack modeling.

A mission thread is defined as the sequence of end-to-end activities and events that takes place to execute a military operation. The MTW is an analysis technique to identify and rank requirements for mission threads [Gagliardi 2013]. An outline of using mission threads to assess risk can be found in *Cyber Security Engineering* [Mead 2016].

While the MTW focuses on specific scenarios, these can be oversimplifications. The SERA framework [Woody 2016] is a more holistic approach to understanding the system in its operational or production environment using multiple models to represent aspects of the system that affect system security, including critical data, access paths, and threat outcomes.

Assurance cases [Goodenough 2012] have been found to be a useful tool to reason about safety and potentially security. In particular, misuse or abuse cases have been applied to understanding security requirements [Sindre 2005, McDermott 1999].

For analysis, the activities include threat modeling and attack surface analyses. Support tools for these include diagraming, issue tracking, support for STRIDE, and automated feedback. Simple tools include white boarding; software support tools include Microsoft's SDL Threat Modeling tool[4] and SeaMonster.[5]

After these, human review is a primary support activity in requirements management. Structured techniques for listing and prioritizing non-functional requirements include business threads (mission threads) and Quality Attribute Workshops [Barbacci 2003]. It is also possible to specify a safer language or use of a secure operating system.

Requirements work is most likely to be substantial on new development, but it can also be relevant during enhancement, migration, and reengineering. *SOAR* has few tools to apply during this stage. Large projects should have a dedicated management tool, and at a minimum, all requirements should be human-reviewed. New development, integration of components, and acquisition can all include selection of secure components. This can be addressed during architecture activities.

---

[4] See http://www.microsoft.com/security/sdl/adopt/threatmodeling.aspx0.

[5] See https://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx.

Any technical objective may be addressed during requirements. Explicitly documenting the requirements is important for identifying ways to achieve specific technical objectives.

*Table 4: Tools and Techniques for Requirements*

| Tools for Requirements | Comments |
|---|---|
| Threat Modeling | Includes use of attack trees, abuse cases, surface analysis, data flow diagrams and attack patterns. Structured approaches include PASTA, LINDUN, CVSS, STRIDE, Attack Trees, Persona non Grata, Security Cards, Quantitative TMM, Trike, VAST Modeling, Octave, and hTMM [Shevchenko 2018a] |
| Mission Thread Workshop | An end-to-end analysis of the system; often a first step for software architecture [Gagliardi 2013] |
| Quality Attribute Workshop | Use to refine and prioritize the security of the attributes most critical to the domain, (e.g., integrity, availability, recoverability) [Barbacci 2003] |
| Security Engineering Risk Analysis (SERA) Framework | A framework for explicitly considering and documenting risks [Alberts 2014] |
| Assurance Cases | A formal approach to document the claims and evidence that the system is sufficiently safe |
| Abuse Cases | Similar to a use case, model interactions with the system that could be harmful |
| Manual Review | Always recommended |
| Negative Test Cases | Can help line-of-code and branch coverage. This can be difficult and expensive without source code. |

#### 6.3.1.2 Architectural and Design Considerations

Several technical objectives (TOs) have particular importance in architecture and design. Obviously TO 1 (ensure quality of the design) applies (see page 77 for a full description of technical objectives). TO 3 (ensure authentication and access control) needs to be addressed in the design. TO 5 (avoiding malicious logic) avoids later problems by addressing misuse or abuse cases or reducing the attack surface.

The attack surface is outlined in the *SOAR* [Wheeler 2016] and elaborated in *Threat Modeling* [Swiderski 2009]. Many of the CERT top practices listed in Section 2.4 are used in the architecture and design phase, including architect and design for security policies, keep it simple, default deny, adhere to the principle of least privilege, sanitize data sent to other systems, practice defense in depth, and validate input.

Tools and techniques that connect requirements to design, record design decisions, and help you reason about alternative design approaches are useful during this stage. Architectural designs often begin with the outputs from the MTW. The Architectural Tradeoff Analysis Method (ATAM) workshop is a structured approach to evaluate how architectural approaches satisfy the requirements [Kazman 2000].

Attack modeling can evaluate the design against potential attacks. A source of common attacks is available from the Common Attack Pattern Enumeration and Classification (CAPEC) [MITRE 2018a]. One threat modeling approach to consider, STRIDE, has been used at Microsoft [Microsoft 2002]. Attack trees are among the techniques described in *Toward a Secure System Engineering Methodology* [Salter 1998].

The Architecture Analysis and Design Language (AADL) provides a precise set of semantics for describing the system at the architectural level and generating code from the design, and it has been extended to apply to cyber-physical systems [Ellison 2015].

Finally, assurance cases [Goodenough 2012] provide both a structured approach and a formal notation to document the body of evidence supporting the claims that a system has certain properties in a given environment. That is, the assurance cases document the claims, the evidence, and the assumptions that support a formal argument justifying a level of confidence that the system will have the critical properties. The form is similar to a legal argument and can be used both for communication, documentation and review. Often applied to safety-critical systems, the technique also supports a holistic analysis of security threates, risks, countermeasures, and mitigation [Lautieri 2005].

Architectural design is most active during initial project development; however, it is essential that the architecture is not compromised during maintenance or a product is not repurposed in a way that is inconsistent with its design. Architectural design activities and reviews should, therefore, be invoked during all system lifecycle stages.

Initial architecture usually comes after some core functionality and key non-functional (quality) attributes have been baselined. Threat modeling will identify the critical assets: data to be secured from destruction or exfiltration, responsiveness, downtime limits, and so forth (for example, see the STRIDE threat model[6]).

The *SOAR* has very limited explicit coverage of architecture. Relevant tools include attack modeling and, for legacy systems requirements, aware knowledge extraction. These typically address TOs 1, 3, 4, and 5 (code quality, access control, unintended weaknesses, and malicious logic). Attack surface modeling requires at least an initial system architecture and can, therefore, be applied after an architectural design has been prepared. Architectural design can address TOs 1, 2, 3, 4, and 5 (code quality, known vulnerabilities, access control, unintended weaknesses, and malicious logic) by specifying use of secure libraries, use of a secure operating system, and safer languages. TO 9 (secure configuration) can be addressed by specifying digital signature verification.

Human review is also recommended. A structured review technique for architecture is the ATAM [Kazman 2000].

*Table 5: Tools and Techniques for Architectural Design*

| Tools for Architecture | Comments |
|---|---|
| Attack surface modeling | Includes use of attack trees, abuse cases, and surface analysis and attack patterns. |
| Architecture Tradeoff Analysis Method (ATAM) | Used to analyze the architecture design |
| Architecture Analysis and Design Language (AADL) | AADL provides formal design that can be verified and can often support automatic generation of code |
| System Theoretic Accident Model and Process (STAMP) | A causality model that enforces safety and security constraint |

[6] https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx

| Tools for Architecture | Comments |
|---|---|
| Systems Theoretic Process Analysis (STPA) | Hazard analysis based on STAMP |

#### 6.3.1.3 Code and Construction Considerations

A guiding principle is to avoid or remove the most common types of vulnerabilities; this is the so-called 80-20 rule (or Pareto principle). Since these are the most common vulnerabilities, they are ones you are likely to have put into your code unless you do something about it. In practice, developers need to know the common types of vulnerabilities (e.g., via CWE/SANS top 25 and the OWASP top 10) and select at least one countermeasure for each that is relevant to their domain.

Detailed design and code require attention to TO 1 (quality), TO 2 (known vulnerabilities in packages), TO 4 (unintended weaknesses), TO 5 (injection of malicious code), TO 6 (anti-tampering), and TO 7 (tool-inserted weaknesses). TO 9 (secure configuration) should also be applied to the development environment and outputs.

A 2004 study determined that 64% of the vulnerabilities in the National Vulnerability Database were due to programming errors, and 51% of those were due to classic errors like buffer overflows, integer overflow, cross scripting, and injection [Heffley 2004]. A review of the top 25 CWE suggests that the situation was unchanged in 2016 [Woody 2014]. Table 6 describes a large number of tools to consider in code and construction phases.

*Table 6: Minimal Tool Sets for Code Through Test*

| Tool for Code | Comments |
|---|---|
| Secure coding standards | Secure coding standards are available for a number of languages to provide safer language subsets that avoid common weaknesses and undefined behavior. |
| Safer languages | Some languages, such as SPARK [Barnes 2009], are more resistant to weaknesses. |
| Secure libraries or components | These can also be investigated during architecture, but are often identified during the implementation. |
| Manual review | Usually recommended. At a minimum, review all new and changed code. Can be very effective for TO 1, 2, 3, 4, and 5. |
| IEEE Inspection | At a minimum, inspect all new and changed code. Can be very effective for TO 1, 2, 3, 4, and 5 but requires training. |
| Compiler warning flags | If a compiled language, establish a standard. Partially addresses TO 1. |
| Source code quality analyzer | Usually recommended; partially addresses TO 1 and 3. See the *SOAR* for additional comments. |
| Source code weakness analyzer | Usually recommended. Different tools often provide different warnings; some projects use more than one tool. Partially addresses TO 1, 2, 3, and 4. |
| Version control | Always recommended. It is essential to maintain traceability to component versions and identify all changes. |
| Debugger | Debuggers are almost always available for systems. |
| Development of negative test cases | Always recommended for security; includes bad input. Partially addresses TO 1, 2, 3, and 4. |
| Test coverage analyzer | Can help line-of-code and branch coverage. This can be difficult and expensive without source code. |

| Tool for Code | Comments |
|---|---|
| Origin analysis | Verifies the provenance of components and helps avoid known CVE. |
| Formal correctness by construction | When used with safe languages such as SPARK, properties of the software can be formally proven. |
| Obfuscation detection | Detects deliberate attempts to hide source code functionality. |

### Secure Coding Resources

*Table 7: Resource List for Secure Coding*

| Resource | Description |
|---|---|
| Intellipedia at Intelink (https://intellipedia.intelink.gov/wiki/Secure_Coding_Guidelines) | A collection of wikis that include secure coding standards |
| *Secure Programming HOWTO* (https://www.dwheeler.com/secure-programs/) | A free online book that provides a set of design and implementation guidelines for writing secure programs |
| *Building Secure Software: How to Avoid Security Problems the Right Way* [Viega 2002] | A book describing a proactive approach to computer security |
| Open Web Application Security Project (OWASP) | An online community with articles, methodologies, documentation, tools, and technologies related to web application security |
| *OWASP Secure Coding Practices Quick Reference Guide* | A short, technology-agnostic set of software security coding practices in checklist format |
| *OWASP Secure Coding Cheat Sheet* | A list of acceptable secure coding practices |

#### 6.3.1.4 Build and Integration Test Considerations

The build system should support multiple technical objectives (see Appendix D). They include TO 1 (quality), TO 2 (known CVE), TO 5 (known malware), TO 7 (tool-inserted weaknesses), and TO 9 (secure configuration).

The integration activity combines and tests the software components as a group. The product is typically binary or byte code. The test assures that the components are complete and that the interfaces are sound. A key practice is to automate as much of the tool chain as practicable. Automation can also capture counts and dispositions of the findings. These can be used to assure that defective code is not checked in and to measure quality improvement as the software proceeds through the development activities.

*Table 8: Tools to Consider for Integration Test*

| Tools for integration | Comments |
|---|---|
| Fuzz test | These will perform a type of negative testing on inputs. Some may be enhanced with the source code; others may require sample inputs.<br>These should be strongly considered for databases, web applications, and C/C++ codes that take input.<br>While we do not recommend fuzz testing as the only method to satisfy TOs 1, 2, 3, and 4, it supports these objectives. Fuzz testing can also verify the effectiveness of other vulnerability removal techniques and can be applied to binary. |

| Tools for integration | Comments |
|---|---|
| Negative testing | Tests that should fail, in particular those related to security, should be included in any test suite. For example, test to ensure that invalid cryptographic certificates (such as basic self-signed certificates), empty usernames, and empty passwords are rejected. |
| Test coverage analyzer | This can help line-of-code and branch coverage. Usually the source code is required to actually fix coverage problems. Without source code, this is more likely to be an acceptance tool. A possible use would be to identify components without source code that should be replaced. |
| Virus scanner | This can find some binary or bytecode patterns and requires little cost or effort. |
| Bytecode/binary weakness analyzer | These can identify unintentional known like weaknesses. |

There are a wide variety of fuzzing tools, some of which require source code. If the product includes binary or byte code components, these components should be checked.

*Table 9: Tools to Use with Binary or Byte Code Libraries*

| Tools for Integration | Comments |
|---|---|
| Origin analyzer | Verifies pedigree and can counter known like weaknesses and vulnerabilities (TOs 2 and 4). Also called a software composition analysis tool. This can be applied to open source and some closed source, and some analyzers apply to binaries or bytecode. |
| Bytecode/binary weakness analyzer | These can identify unintentional known like weaknesses. |

#### 6.3.1.5 System and Acceptance Test Considerations

The test criteria can be broadly interpreted to apply to all the listed technical objectives. Particular emphasis, however, will usually be applied to verifying and validating TO 1 (quality), TO 2 (known CVEs), TO 3 (authenticator and access controls), TO 4 (unintentional weaknesses), TO 5 (resistance to known malware), and TO 6 (anti-tampering).

*Table 10: Tools to Consider for System and Acceptance Test*

| Tools for Test | Comments |
|---|---|
| Negative testing | Tests that should fail should be included in any test suite. |
| Fuzz test | These will perform a type of negative testing on inputs. Some may be enhanced with the source code; others require sample inputs.<br>These should be strongly considered for databases, web applications, and C/C++ codes that take input.<br>While we do not recommend fuzz testing as the primary method to satisfy TOs 1, 2, 3, and 4, it is useful to verify the effectiveness of other vulnerability removal techniques and can be applied to binary. |
| Test coverage analyzer | This can be applied to unit or system test. Analysis of coverage can be expensive without source code, so this is best applied by the development organization. |

#### 6.3.1.6 Deployment and Operations Considerations

Deployment emphasizes TO 8 (secure delivery) and TO 9 (secure configuration). However, as vulnerabilities are discovered, TO 2 (known CVEs) should be monitored.

The following tools and techniques can be useful in operations:

- logging systems
- security information and event management (SIEM)
- hardening tools/scripts
- tracking sensitive data
- network sniffer (or packet analyzer) to monitor network traffic
- network scanners to actively interact with network nodes and components
- traditional virus checkers
- inter-application flow analysis to examine the control or data flows among a set of applications and verify that security policy is not violated
- comparing binary or bytecode to the permission manifest
- obfuscated binary/bytecode detection
- secured operating system
- origin analysis to verify code pedigree and version
- digital signature verification to verify software has not been tampered with
- configuration checker to verify settings are proper
- vulnerability scanner to verify the system host configuration
- host application interface scanners to enumerate application interfaces
- web application vulnerability scanner
- web services scanner
- database vulnerability scanner
- digital forensics
- intrusion detection systems
- automated monitored execution
- firewall

*Table 11: Minimal Tool Sets for Deployment at the Application Layer*

| Tools and Techniques | Comments |
|---|---|
| Hardening tools and scripts | Are often available for OTS systems and can be very effective for TOs 2, 3, and 8. |
| Configuration checking | Supports TO 8 (provide secure delivery). |
| Digital signature verification | Must be implemented during development. Supports TO 8 (provides secure delivery). |
| Applicable or server scanners<br>• web application scanner<br>• web services scanner<br>• database scanner<br>• host-based vulnerability scanner | If TO 9 (provide secure configuration) is included, scanners provide an automated way to verify that the configuration is correct. They often include fuzzing tools. |
| Execute and compare with application manifest | Verifies application permission is consistent with specification. |

*Table 12: Tool Sets for Deployment Above the Application Layer*

| Tools and Techniques | Comments |
|---|---|
| Intrusion detection systems | Can be very effective for TOs 2 and 4. |

| Tools and Techniques | Comments |
|---|---|
| Network scanners<br>• network sniffer<br>• network vulnerability scanner | One or more of these tools are applicable in networked environments, where databases are used, or where configuration cannot be assured with manual spot checks. |
| Firewalls<br>Host-based vulnerability scanner<br>Network intrusion detection | These apply when the software is in use in the field. They involve actively monitoring how the product is being used, attacked, compromised, or changed. |
| Virus scanner | Virus scanners are typically inexpensive and should be used wherever appropriate. They are very effective for TO 2 and partially address TO 5. |

#### 6.3.1.1 Maintenance Considerations

Tools and techniques applicable during maintenance include those used in all other lifecycle stages. Some that can be of particular interest are listed in Table 13.

*Table 13: Tools and Techniques in Maintenance*

| Tools and Techniques | Comments |
|---|---|
| Source code knowledge extractors | Helpful to understand legacy systems |
| Threat models | Should be continually maintained |
| CVE reports | Monitors vulnerability reports |
| Binary/bytecode dissembler | Verifies no malicious changes have been introduced |
| Rebuild to compare | Ensures that the build includes only what is intended and has not changed |

### 6.3.2 Special Lifecycle Considerations

This section provides some considerations for specific product lifecycle stages.

#### 6.3.2.1 New Development

In addition to the tools and techniques described in Section 6.3.1, new products introduce opportunities to

- select safer languages
- select secure libraries
- inspect architecture, requirements, and designs
- establish security requirements

#### 6.3.2.2 System Reengineering

Reengineered systems typically have specified functional requirements and may have specified interfaces to other systems. The trigger for redevelopment is usually a quality attribute such as maintainability, extensibility, performance, or security. Safer languages and secure libraries should be used where practical. In addition, source code knowledge extractors can be helpful if legacy source code is available. As with new development, use this opportunity to

- select safer languages
- select secure libraries
- inspect architecture, requirements, and designs

- establish security requirements

When updating systems or systems of systems, special attention must be paid to component trust relationships. Not only interfaces, but also assumptions, can change. It is not sufficient to verify only the individual components. The component or component system interactions and assumptions must be verified and tested.

#### 6.3.2.3 Maintenance and Bug Fixes

Many older systems are in maintenance, with bug fixes or minor platform alterations making up most of the changes. Often components and libraries (for example COTS, GOTS, or OSS) with newly discovered vulnerabilities must have patches installed. At a minimum, a risk and remediation cost assessment should be performed on the system. Program management will need to determine if the system can continue to operate, can be operated only after some fixes are applied, or must be replaced.

The assessment should begin with threat modeling and attack surface analysis. Known threats should be examined using virus checks, origin analysis, or web tools, as appropriate to the system. A baseline should be performed using source code quality and weakness analyzers.

The activities during enhancement typically include limited new code, rebuilding, test, and deployment. These should be treated as similar to developing new code. There is typically no opportunity to select the programming language and limited opportunity to select secure components. Open source or commercial source and components are sometimes included. The recommendations are similar to those for new code, but they come with the warning that a large number of issues may be identified.

The most defective modules can often be identified by bug history and warnings from static analysis. Bug tracking systems and static analysis warnings can be used to target components for manual inspection. Always perform a manual review of the changes and the changed module. Defects tend to cluster, and the defective module should be considered to be at risk (TOs 1, 4, 5).

The IEEE suggests formal inspection of changes and the changed module because roughly 15% of defect fixes introduce a new defect, and only 50% of these defects are likely caught by test (TOs 1, 4, 5). This also applies to the following:

- adopting and enforcing standard compiler warnings (TOs 1, 4)
- using a static code quality and/or weakness analyzer (TOs 1, 4)

It is possible that a weakness analyzer will produce a very large number of warnings. First address warnings in any component that has been changed. The engineer must report the technical aspects of risk and cost estimates for remediation to program management for business risk management determination.

TO 2 counters known vulnerabilities. If COTS, GOTS, or OSS is used, the CVE should be reviewed for new vulnerabilities. Origin analysis may be helpful at identifying packages known to have a vulnerability.

Test coverage analysis to address TOs 1 and 4:

- Introduce negative tests to exercise unintended inputs where appropriate.

- Fuzz test can be moderately effective for TOs 1 and 4 and has identified a substantial number of issues in legacy and OSS codes.
- If external components are included, consider binary/bytecode scanners for external packages.

## 6.4 Getting Started with Secure Development

At this time, there is little empirically grounded data to make rigorous economic decisions on the cost and effectiveness of different tools and tool types. There is good evidence that static and dynamic analysis is operationally cost effective and simple to implement; these should be used wherever practicable. Additional recommendations are based on observations of development practice.

Begin by determining which class of either manual or automated security testing (AST) is appropriate for your application. Keep these concepts in mind to begin selecting appropriate tools or countermeasures:

- The key is not “secure vs. insecure”; it is risk reduction.
- There is no one tool or countermeasure that will that will solve the problem.
- There are factors to help determine which manual techniques and classes of AST tools can be helpful.
- Begin with simple practices and incrementally improve.

Security is not binary: the objective is to manage risk and exposure. There are many potential sources of vulnerability, and different techniques have different effectiveness against weakness or vulnerability types. Moreover, while automated tools can be very effective against specific weaknesses, they are seldom perfect against all weaknesses, even of a given type. No one technique, tool, or tool class will make the product free from vulnerabilities. However, each tool can reduce risk and exposure. Begin with the lifecycle and development stages to determine which tools and techniques are available (see the *SOAR* matrix and Section 6.3.1 of this report); then consider a subset of the additional contextual factors:

- in-house vs. third-party development
- source code availability
- experience and capability of the staff
- maturity of the tool set being considered
- compliance with contract or regulations
- development approach
- target platform
- maturity stage of the product
- opportunity to integrate tools/techniques into development

In the following paragraphs, we discuss these factors in isolation, and in some combinations.

### In-House vs. Third-Party Development

If the application is written largely in-house, use static source code analysis. The tool set is mature, robust, effective, and operationally inexpensive, and it can be integrated into most

development environments. High-quality commercial tools are more effective for common weaknesses and may have usability advantages.

For commercial or open source components, origin analysis or software composition analysis is a near must. Origin analysis should, at a minimum, identify components with known vulnerabilities.

For other third-party development, dynamic analysis is a reasonable first choice. If source code is available, static analysis can be used, but the usefulness is limited without developer evaluation and inclusion in the build process.

While these are good choices if authorship is the only thing known about an application, all the decision factors together will influence the final decision.

### Source Code Availability

Source code provides a lot of flexibility. Static code analysis tools are the most robust and should be used whenever practical. Some static binary analysis tools also require source code and the build system. When source code is available, use static source analysis tools or a combination of static and dynamic tools. If source code is not available, use dynamic analysis tools.

If the application was written by a third party and the source code is not available, consider using fuzzing and negative testing tools in addition to traditional dynamic analysis tools.

If the application was written by a third party and the source code is available, consider running build and compare tools.

### Third-Party Components

Third-party development can include contracted development, proprietary components or libraries, and open source components. Use source composition or origin analysis first. Often this will identify components or libraries that were not thought to be in the application. It is hard to overstate the risk associated with using obsolete or unpatched components containing known vulnerabilities.

If there are some, but few, third-party components, apply static analysis first and supplement with composition analysis. If the product is largely written by a third party, apply software composition analysis and supplement with dynamic analysis.

### Application Maturity

Static analysis is almost always a cost-effective means to find and quickly repair certain types of weaknesses. This is true regardless of the product age. Nonetheless, large bodies of legacy source code can produce an unmanageably large volume of warnings. The advantage is that individual warnings can usually be fixed very quickly. Dynamic testing tools are more likely to find potentially exploitable weaknesses, but the defects are likely to be more difficult to find and repair.

### Development Approach

Software development varies by release schedule, granularity of increments, frequency of code commits, and frequency of build. The shorter the development cycle, the more important it

becomes to integrate secure development tools into the development environment. Static analysis tools can be integrated into unit test builds and source code commits. Static, dynamic, and hybrid tools can be effectively integrated into system builds. System builds are more likely to pass test if static analysis is performed prior to system test. Dynamic test tools can be used either to find more defects prior to test or to measure the effectiveness of test coverage. If test uses specialized and expensive equipment, or a minimum number of test cycles is important, perform dynamic test prior to test.

If builds are frequent and third-party components are present, it is a good idea to integrate software composition analysis into the build process.

### Target Platform

If the application is local, such as batch or desktop, static analysis is likely the best initial choice. On the other hand, because of the attack surface presented, Internet-facing applications strongly suggest the use of dynamic analysis.

Mobile applications present specific attack-surface problems that should be specifically addressed with mobile application security tools (MAST).

An application built or run on the cloud should employ tools for application security testing as a service (ASTaaS).

Many applications written in-house include open source components to support both web and mobile versions. This presence of source code suggests using static analysis. Internet and mobile indicate using both dynamic analysis and mobile application tools. Origin analysis should be used whenever open source components are included to ensure that known vulnerabilities are not present.

### Integration Level

Integration level refers to the practicability of adding tools into the development process and the degree to which the process can be automated. Static analysis and origin analysis are usually straightforward to integrate into the build. Static analysis can be run effectively at any time in the development cycle, but it is most effective when the results can be promptly addressed by the developers.

Tools typically integrated later in the development cycle include dynamic analysis, fuzzing, and negative testing.

Threat modeling and attack surface modeling are challenging to integrate into development, but should occur early. In environments where requirements change rapidly, threat modeling and attack surface modeling should be revisited frequently as part of the change process.

### Compliance

Statutory, regulatory, policy, or contract compliance may dictate certain tools. Examples include but are not limited to the Health Insurance Portability and Accountability Act (HIPAA), the Federal Information Security Management Act (FISMA), the Sarbanes–Oxley Act (SOX), and the Defense Acquisition Regulations System (DFARS). Forms of compliance vary: some require a

specific tool or tool class, while others require compliance with an industry standard (e.g., Control Objectives for Information and Related Technologies [COBIT], ISO, and the Risk Management Framework [RMF]).

Database security scanners are often helpful for the storage and protection of data. Correlation and Application Security Testing Orchestration (ASTRO) tools help with reporting.

#### Maturity of the Tool Set

Initially most tooling will be done using static code analysis, static binary/byte code analysis, dynamic analysis, origin analysis, and database security scanning. These are the most mature of the AST tools that address most common weaknesses. These tools typically have a low barrier to initial use. After gaining proficiency and experience, consider adding architecturally appropriate tools to support MAST, ASTaaS, and interactive application security testing/hybrid tools.

### 6.4.1 Tool Type Factors Summary

Examining each factor allows you to build a list of tool types to consider. While some factors may lead toward a certain type of tool, other factors may lead away from that type.

Ideally you will implement a combination of tools. Static analysis and dynamic analysis are complimentary, and origin analysis should be used whenever third-party components are present. Use interactive security testing and hybrid tools if needed to get the most coverage. When budget or experience allow only one or two tool types to be considered, use the decision factors to prioritize.

The combined use of static analysis, dynamic analysis, and origin analysis is a good starting point and provides experience to understand how to expand use to MAST, interactive application security testing (IAST), and ASTaaS.

### 6.4.2 Considerations for Selecting Specific Tools

#### Existing Development Technology

Select tools that are compatible with the intended programming languages. Some tools support only a single language, while others support groups of languages. Where possible, select tools that can be integrated with existing IDE and build systems.

#### Technical Objectives

The technical objectives may lead toward specific actions. For example, the technical objectives may include avoiding known vulnerabilities (CVEs), protecting against SQL injection, avoiding buffer overflows, preventing cross-site scripting, verifying password management, and so forth. License adherence would strongly encourage a compatible origin analysis tool.

#### Cost and Human Resources

Many of the tools save development time by finding defects and vulnerabilities early. Nonetheless, they require acquisition, training, integration into the development environment, and calibration to the local environment.

# 7 Analyzing and Responding to Software Assurance Findings

Many findings, especially those that are low cost in terms of time and effort, should simply be addressed. These include compiler warnings and other warnings that can be addressed at the code and detailed design level.

Not all findings, however, are true positives, and not all true positives must be addressed. The developers need to be capable of communicating the costs and risks to program management. To do so, they must be aware of what management needs to know and be capable of speaking that language. This section describes the cost and benefit decisions relevant to the developer.

## 7.1 Introduction to Risk

The goal of a software assurance risk assessment is to identify weaknesses or known vulnerabilities in the software product. Risk involves two dimensions. The first dimension is the likelihood of exploitation; for example, if the weakness is not accessible through the attack surface, the likelihood may be as low as zero. The second dimension is the severity of the consequences if the weakness is exploited. This can range from minimal consequences to loss of mission or even loss of life. The combined likelihood and consequence provide the risk exposure. This risk must be balanced against the costs of remediation or the opportunity costs of not using the product.

## 7.2 The Mission Thread

Risk begins with the threat to the mission. The mission helps to determine the context. Woody and Mead approach this from the mission thread [Mead 2016]. Mission thread is the U.S. military term for workflow, defined as a collection of interrelated work tasks that achieves a specific result [Leveson 2004]. The mission thread is, therefore, a sequence of end-to-end activities and events that takes place to accomplish the execution of a military operation. An analysis technique for mission threads is the Mission Thread Workshop [Gagliardi 2013].

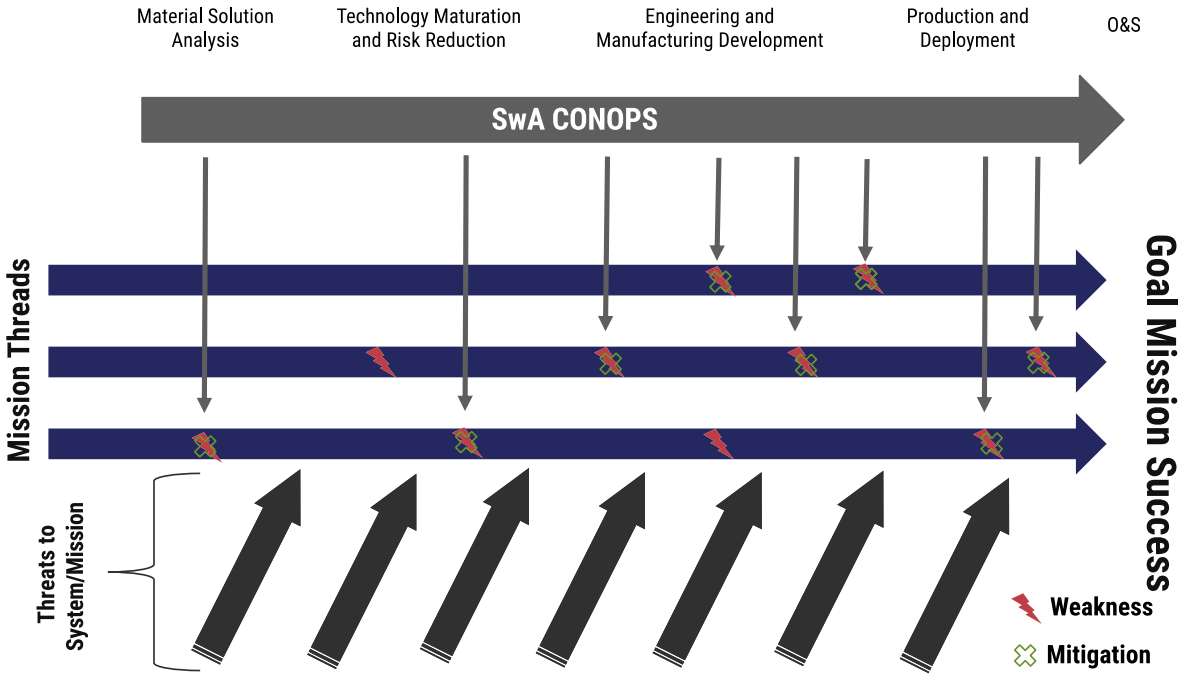
An outline of using mission threads to assess risk can be found in *Cyber Security Engineering* [Mead 2016] and the SERA Framework [Alberts 2014]. A summary of the steps are as follows:

1. Establish the operational context.
2. Identify the risk.
3. Analyze the risk.
4. Develop a control plan.

## 7.3 CONOPS

A concept of operations (CONOPS) is a short but clear description, in text and/or graphics, of what needs to be accomplished and how it will be done. A software assurance CONOPS identifies the behaviors, activities, tools, techniques, and countermeasures focused on identifying and mitigating threats to the mission. The CONOPS, mission threads, and product lifecycle stages

perspectives can be combined in a single overview as in Figure 12. The overall goal of the system and its interfacing systems is operational success. Each mission thread has weaknesses that can be addressed by activities in the SwA CONOPS. Since threats do not go away, there are risks in not adequately addressing weaknesses. This view can help analyze the risks.




derived with permission from a diagram originally developed by DASD(SE)

*Figure 12: Conceptual View – Software Assurance Mission Success*

## 7.4 Risk Analysis

Not all findings must be remediated immediately. Where the cost is low (e.g., remediation will not affect schedule, commitments, or staffing level), "just fix it" is a good policy. Otherwise, management requires information about the risk exposure and the cost of alternatives. Developers should provide management with the expected cost in effort and delay, and the risk introduced by not mitigating a finding. There are several ways to analyze the weakness findings from security scanning tools and rank them. These include the following:

- weakness types (i.e., the CWE)
- groupings by the Defense Information Systems Agency
- Common Vulnerability Scoring System (CVSS)[7]

The CVSS provides a calculation to quantify the severity of a vulnerability. CVSS includes a base score and modifiers for environmental and temporal factors. One component of the base score is based on the National Vulnerability Database (NVD) severity rating. Other components include exploitability consequences on confidentiality, integrity, and availability.

[7] See https://www.first.org/cvss/.

## 7.5 Controlling the Risk

Controlling risks involves one or more of the following:

- Accept – If a risk occurs, its consequences will be tolerated; no proactive action to address the risk will be taken.
- Transfer – Risk mitigation is shifted to another party (e.g., through insurance or outsourcing).
- Mitigate – Actions are implemented in an attempt to reduce or contain a risk.
- Avoid – Activities are restructured to eliminate the possibility of a risk occurring; this includes changing the workflow and remediating the weakness.

A very low-likelihood or low-consequence risk might be accepted and the rationale documented.

To remediate a risk means to change the software to directly remove the source of the risk. Mitigation is to take other steps that do not remove the vulnerability, but reduce the opportunity for damage. If the cost of remediation is below some threshold determined by program management, the remediation should be implemented. If the remediation cost is above that threshold, the analyst and developers should provide program management with the cost estimate for remediation, along with risk analysis and other control options.

# 8 Software Assurance During Sustainment[8]

Software spends most of its useful life in maintenance or sustainment. During this time much can change, including the following:

1. The attack surface might change because of changes to the mission or adversary capability.
2. The operational environment may change.
3. Hardware platforms may be replaced.
4. Software packages and components may be revised.
5. New vulnerabilities in components can be discovered in existing components.
6. Development staff will turn over.
7. Legacy issues may arise.

As stated in *Cyber Security Engineering* [Mead 2016],

> *Every component of the software system and its interfaces must be operated and sustained with organizational risk in mind. The planning and execution of the response is a strategic requirement, which brings the absolute requirement for comprehensive lifecycle protection processes into the discussion. Much of the activity is similar to development.*

## 8.1 Preparing for Sustainment

Software sustainment activities are fundamentally the same as software development activities, only the name has changed. The decisions made during development and reasons for those decisions should be documented and provided to the sustainers for evaluation and review. Ideally, the tools, countermeasures, and development environment used during development should continue to be available during sustainment. This not only preserves organizational memory but also provides technical continuity.

Nonetheless, special attention is required to changing conditions, evolving code bases, and changing legacy components.

## 8.2 Steps for Assurance in Sustainment

This set of steps applies when software is in use in the field and involves actively monitoring how the product is being used, attacked, compromised, or changed.

1. Identify – Develop the organizational understanding to manage cybersecurity risks to systems, assets, data, and capabilities.
2. Detect – Develop and implement the appropriate activities to identify the occurrence of a cybersecurity event.
3. Recognize and respond – Monitor the threat and take action when it is detected.

[8] The material in this section is drawn primarily from [NIST 2014].

4. Resist – Implement protection measures to reduce vulnerability to the threat and minimize the consequences that might occur.
5. Recover – Have a plan to recover from the risk if the consequences or losses are realized and restore capabilities or services that were impaired by a cybersecurity event.

### Identify Work Products in Sustainment

The list of work products includes (but is not necessarily limited to) the following:

- threat models
- architecture and design
- source code
- libraries
- COTS/GOTS components
- test cases
- user instructions
- PPPs

### Detect and Monitor Threats and Attacks

Detection includes continuous monitoring, noticing anomalies and events, and generally having detection processes in place. Monitoring can help to accomplish the following:

- benchmark normal organization and system behavior
- scan for malicious emails
- perform network monitoring
- scan malware introduced into the system
- identify anomalous system behavior
- identify events, signatures, and so forth to alert the organization about known malicious behavior

### Recovery

The topic of recovery from cybersecurity events is addressed in the NIST publication *Guide for Cybersecurity Event Recovery* [Bartock 2016]. While the planning, prioritization, and resource allocation aspects are under the control of program management, the development staff is responsible for preparation and implementation. In anticipation of cybersecurity events, plans should incorporate the following:

- recovery guidance and a playbook with major phases, including procedures, stages, and well-defined exit criteria for each stage, such as notification of key stakeholders
- specific technical processes and procedures expected to be used during a recovery
- prepared and documented operational workarounds
- planned and documented details for off-site storage, infrastructure, hardware, and software to be used during a recovery

After recovery, post-action analysis (PAA) and root-cause analysis (RCA) should be performed. Some key recommendations include the following:

- Record enough information to support RCA and expand documentation.
- Validate recovery capabilities by soliciting input from individuals with recovery responsibilities and conducting exercises and tests.
- Use the PAA and RCA to identify weaknesses in the organization's technologies, processes, and people to improve the organization's security posture.

## 8.3 Evolving the Threat Model

### 8.3.1 Finding and Fixing Vulnerabilities

Threats evolve over time as new weaknesses and vulnerabilities are discovered, the software is used in new ways or in different environments, and as motivations for attackers change. In addition to monitoring the software in use, preemptively remove vulnerabilities before they are exploited. To recognize change, however, it is necessary to have a baseline. To assure continuity, baseline identification and mitigation processes should be maintained from development whenever practicable.

To prioritize issuing patches, refer to the CVSS.

### 8.3.2 Tool Considerations in Sustainment

Known vulnerabilities from the NVD and CVE often reside in components obtained from open source or commercial libraries. Origin analysis should be applied frequently to help assure that components with known vulnerabilities are detected. Builds should employ current static and binary analysis. The provenance of the distribution should be maintained with adequate version control. Some tools are available to assist with patch deployment.

### 8.3.3 Maintaining the Processes from Development

The identification and mitigation processes used in development provide institutional memory of how the tools, techniques, and countermeasures to threats are engineered into the software. Where practicable, these processes should be maintained and enhanced rather than discarded.

# 9 Software Assurance Considerations for Acquisition

In software acquisition, most or all of the product is produced by third-party developers or integrators. The separation between the developer and the customer requires special attention to assure that risks are identified and managed. This separation introduces challenges, including the communication of needs, requirements, and status. It is usually necessary for requirements and agreements to be more formally described in contracts and other documents. This section summarizes some of the requirements that acquisition demands of the supplier.

Effective security requires the integration of security into software acquisition. Risk management includes the identification of possible threats and vulnerabilities within the system, along with ways to accept or address them. The Risk Management Framework [NIST 2010] and its artifacts, including the PPP and the Software Assurance Plan, are the responsibility of the software acquirer. The items included, however, must be negotiated and contracted with the development organizations.

## 9.1 Security Requirements in Acquisition

Threat and attack surface models are key to requirements and product architecture. Developing and documenting the agreed prioritization of security activities is critical during acquisition. The SQUARE security requirements approach for development [ISO/IEC 2011] has been adapted for this use and is called A-SQUARE for COTS Acquisition [Mani 2014].

## 9.2 Development Tools and Techniques

Although acquisition contracts should not micromanage development practice, acquirers can and should set expectations and impose constraints. The use of tools to enhance or verify security can be negotiated and should be tracked. Furthermore, acceptance may specify criteria that can be objectively met by using specific tools. The role of tools in assurance must be included in the PPP and Software Assurance Plan. Tool selection influences acquisition costs because security tools and techniques not only affect the product but also have costs associated with tool acquisition, training, and use. At a minimum, an analysis of development tool types should be performed and a specific set of tools (or alternatives) should be agreed to and included in the formal risk management documentation.

The development organization should use validated secure coding standards where available and report on verification of the implementation. The acquiring organization may insist that certain tools are used (i.e., well-known commercial static analysis tools), that warnings are set to certain thresholds, or that all findings are addressed.

## 9.3 Origin Analysis Tools

Acquiring organizations must take extra care to assure the provenance of the delivered products. Issues can arise anywhere in the supply chain. Origin analysis should be used to assure that all components are recognized and that known vulnerabilities are not included in components.

## 9.4 Verification and Validation Tools

Validation tools can be applied in both development test and acceptance test. The acquiring organization should develop negative test cases from requirements for acceptance. Other appropriate verification tools, such as dynamic testing, should be run by the development organization prior to acceptance test.

## 9.5 Addressing Vulnerabilities, Defects, and Failures

The requirements to dispose of discovered vulnerabilities, defects, or failures should be based on mission risks. Establish requirements and a process for disposing of the issues during development, during test, and after delivery.

## 9.6 Additional Acquisition Resources

*Table 14: Resource List for Acquisition*

| Resource | Description |
|---|---|
| *Capability Maturity Model (CMMI) for Acquisition* [CMMI 2010] | An SEI document that provides guidance for applying CMMI best practices in an acquiring organization |
| *Supply Chain Risk Management Practices for Federal Information Systems and Organizations* [Boyens 2015] | A NIST Special Publication that addresses supply chain issues |

# Appendix A: Regulatory Background

While developers are not required to be fluent in the specifics of the regulatory documents, they are required to be sufficiently aware to not only comply but also to support the compliance and documentation needs of program management. This section provides a brief overview of the laws, standards, regulations, and guidelines relevant to cybersecurity and software assurance for the DoD.

## Mandates

**National Defense Appropriations Act (NDAA) for Fiscal Year 2013, Section 932, Improvements in Assurance of Computer Software Procured by the Department of Defense**, January 2, 2013, states the following:

> *USD (AT&L), in coordination with the DoD CIO shall develop and implement a baseline software assurance policy for the entire lifecycle of covered systems. Such policy shall be included as part of the strategy for trusted defense systems of the Department of Defense.*
>
> In **Section 933** it requests the following:
>
> *(1) A research and development strategy to advance capabilities in software assurance and vulnerability detection.*
>
> *(2) The state-of-the-art of software assurance analysis and test.*
>
> *(3) How the Department might hold contractors liable for software defects or vulnerabilities.*

- **Section 10 of US CODE 2358** states,

  *The Secretary shall develop a coordinated plan for research and development on…computer software and wireless communication applications, including robust security and encryption.*

- **Public Law 113-66 NDAA for Fiscal Year 2014, Section 937, Joint Federated Centers for Trusted Defense Systems for the Department of Defense**, directed the DoD to establish a

  *…federation of capabilities to support the trusted defense system needs of the Department to ensure security in the software and hardware developed, acquired, maintained, and used by the Department.*

  This requirement led to the creation of the Joint Federated Assurance Center (JFAC), which is managed by the Deputy Assistant Secretary of Defense for Systems Engineering (DASD(SE)). JFAC is sponsored by the DASD(SE) and is aligned with JFAC efforts in the area of software assurance.

## International Standards

IEEE/ISO standards provide a common language used in government contracting. This includes standard lifecycle stages and processes in **ISO 12207** [ISO/IEC 2008]. Systems engineering uses the related standard **ISO/IEC/IEEE 15288-2015** [ISO/IEC/IEEE 2015].

The **ISO 15026** series addresses how to discuss and document systems and software assurance [ISO/IEC 2013]. It does not define or measure assurance levels.

Part 1 – Concepts and Vocabulary

Part 2 – Assurance Case

Part 3 – System Integrity Levels

Part 4 – Assurance in the Life Cycle

Information security standards are addressed in **Security and Privacy Controls for Federal Information Systems and Organizations** [NIST 2014].

## DoD Regulations

The overarching regulatory document for DoD cybersecurity is **DoDI 5000.02, Cybersecurity in the Defense Acquisition System, Enclosure 14** [USD(AT&L) 2017]. This document establishes a regulatory requirement for producing a PPP at Milestones A, B, and C. The full-rate production/ full deployment decision (FRP/FDD) references DoDI 5200.39, which requires that PPPs address software assurance vulnerabilities and risk-based remediation strategies. It also requires that PPPs include software assurance as part of vulnerability countermeasures.

There are a number of related regulatory documents:

- **DoDI 5200.44 Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN)**, November 5, 2012
- **DoDI 5200.39 Critical Program Information (CPI) Identification and Protection Within Research, Development, Test, and Evaluation (RDT&E)**, May 28, 2015
- **DoDD 5200.47E Anti-Tamper (AT)**, September 4, 2015
- **USD (AT&L) Memorandum Document Streamlining – Program Protection Plan (PPP)**, July 18, 2011
- **PM 15-001 Deputy Secretary of Defense Policy Memorandum (PM) 15-001, Joint Federated Assurance Center (JFAC) Charter**, February 9, 2015

## DoD Guidelines

In addition to regulations, the DoD has provided guidelines.

- The ***DoD Program Manager's Guidebook for Integrating the Cybersecurity Risk Management Framework (RMF) into the System Acquisition Lifecycle*** emphasizes integrating cybersecurity activities into existing processes, including requirements, systems security engineering, program protection planning, trusted systems and networks analysis, developmental and operational test and evaluation, financial management and cost estimating, and sustainment and disposal.

It provides an outside-in risk management framework for the program manager on integrating cybersecurity activities into the system's acquisition lifecycle, while the guidebook under development provides more of an inside-out engineering perspective of what a program manager needs to know about the engineering-in of software assurance activities. The guidebooks should be compatible with each other and useful to program managers in carrying out their software assurance and cybersecurity risk management responsibilities.

- The ***Engineering for Systems Assurance Guide*** from the National Defense Industrial Association [NDIA 2008] covers program manager and system engineering assurance roles and responsibilities over the system engineering lifecycle. It includes the phases of the DoD Integrated Defense Acquisition, Technology, and Logistics Life Cycle Management Framework as discussed in DoD Directive 5000.01, DoDI 5000.02, the guidance in the *Defense Acquisition Guidebook*, and ISO/IEC 15288 Systems and Software Engineering – Systems Life Cycle processes.
- The ***Defense Acquisition Guidebook*** is designed to complement formal acquisition policy as described in DoD Directive 5000.01 and DoD Instruction 5000.02 by providing the acquisition workforce with discretionary best practice that should be tailored to the needs of each program. The guidebook "is not a rule book or a checklist and does not require specific compliance with the business practice it describes. It is intended to inform thoughtful program planning and facilitate effective program management." In the area of software assurance, Chapters 3 and 9 are of interest. Chapter 3, Systems Engineering, describes standard systems engineering processes and how they apply to the DoD acquisition system. Chapter 9, Program Protection, explains the actions needed to ensure effective program protection planning throughout the acquisition lifecycle.

# Appendix B: Resources

| Resource | Description |
|---|---|
| *Building Secure Software: How to Avoid Security Problems the Right Way* [Viega 2002] | A book describing a proactive approach to computer security |
| *Building Security In Maturity Model (BSIMM)* [McGraw 2017] | A study of existing software security initiatives sponsored by the Department of Homeland Security. It collects the state of professional practice, but does not recommend specific practices. |
| *Capability Maturity Model (CMMI) for Acquisition* [CMMI 2010] | A document providing guidance for applying CMMI best practices in an acquiring organization |
| *Cyber Security Engineering: A Practical Approach* [Mead 2016] | A book in the SEI Series on Software Engineering that provides a reference and tutorial on a broad range of assurance issues and practices |
| *DoD Program Managers' Guide to Software Assurance* | An SEI document that is a companion to this guidebook |
| *Economics of Software Quality* [Jones 2011] | A book by Capers Jones that addresses macroscopic issues |
| *Goal Question Metric (GQM) Paradigm* [Basili 1992] | A premier resource for measurement in software engineering |
| *Goal-Driven Software Measurement — A Guidebook* [Park 1996] | A guidebook to help identify, select, define, and implement software measures to support business goals |
| *Guide for Applying the Risk Management Framework to Federal Information Systems* [NIST 2010] | Guidelines published by NIST for applying the Risk Management Framework to federal information systems |
| *Integrated Measurement and Analysis Framework for Software Security* [Alberts 2010] | This technical report by the SEI provides security metric resources. |
| Intellipedia at Intelink (https://intellipedia.intelink.gov/wiki/Secure_Coding_Guidelines) | A wiki about secure coding guidelines available to individuals with appropriate clearances. |
| Intellipedia at Intelink (https://intellipedia.intelink.gov) | A collection of wikis available to individuals with appropriate clearances. These online resources contain information on various software assurance topics relevant to DoD developers and contractors. Secure coding standards are included. |
| Open Web Application Security Project (OWASP) | An online community with articles, methodologies, documentation, tools, and technologies related to web application security |
| *OWASP Secure Coding Cheat Sheet* (https://www.owasp.org/index.php/Secure_Coding_Cheat_Sheet) | A list of acceptable secure coding practices |

| | |
|---|---|
| *OWASP Secure Coding Practices Quick Reference Guide* (https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide) | A short, technology-agnostic set of software security coding practices in checklist format |
| *Personal Software Process* [Humphrey 1995] | A book by Watts Humphrey that provides a highly useful implementation for measurement in software |
| SAFECode (https://safecode.org) | An industry group "dedicated to increasing trust in information and communications technology products and services through the advancement of effective software assurance methods." |
| *Secure Programming HOWTO* (https://www.dwheeler.com/secure-programs/) | This free online book provides a set of design and implementation guidelines for writing secure programs |
| *Software Quality Metrics Overview* [Kan 2002] | A short introduction to measurement theory and application |
| *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* [Wheeler 2016] | A publication by the Institute for Defense Analyses (IDA) that contains a large volume of information on the types of tools available and contextual factors on how they can affect security |
| *Supply Chain Risk Management Practices for Federal Information Systems and Organizations* [Boyens 2015] | A NIST Special Publication that addresses supply chain issues |

# Appendix C: Tools, Techniques, and Countermeasures Throughout the Lifecycle

*Table 15: Tools, Techniques, and Countermeasures Throughout Lifecycle Processes*

| Tool/Technique | Stakeholder Requirements Definition | Requirements Analysis | Architectural Design | Implementation | Integration | Verification | Transition | Validation | Operation | Maintenance |
|---|---|---|---|---|---|---|---|---|---|---|
| Assurance Case | | x | x | | | | | | | |
| ATAM | | | x | | | | | | | |
| Attack Modeling | | | x | | | | | | | |
| Automated Regression Test | | | | x | | x | | x | | x |
| Binary Weakness Analyzer | | | | x | x | x | | x | | x |
| Binary/Bytecode Disassembler | | | | | x | | | | | x |
| Binary/Bytecode Simple Extractor | | | | x | | | | | | x |
| Bytecode Weakness Analyzer | | | | x | | x | | | | x |
| Compare Binary/Bytecode to Application Permission Manifest | | | | | | | | | x | |
| Configuration Checker | | | | | | | | | x | |
| Coverage Guided Fuzz Tester | | | | | x | x | | | | x |
| Database Scanner | | | | | | | | | x | |
| Debugger | | | | x | x | x | | | | x |
| Development/ Sustainment Version Control | | | | x | x | x | | x | | x |
| Digital Forensics | | | | | | | | | | x |
| Digital Signature Verification | | | | | | | x | | x | |
| Execute and Compare with Application Manifest | | | | | | x | | x | | |
| Fault Injection | | | | | | x | | | | |

| Tool/Technique | Stakeholder Requirements Definition | Requirements Analysis | Architectural Design | Implementation | Integration | Verification | Transition | Validation | Operation | Maintenance |
|---|---|---|---|---|---|---|---|---|---|---|
| Firewall | | | | | | | | | x | |
| Focused Manual Spot Check | | | | x | | x | | | | |
| Forced Path Execution | | | | | | x | | | | |
| Formal Methods/Correct by Construction | | | | x | | | | | | |
| Framework-Based Fuzzer | | | | | | x | | x | | |
| Fuzz Tester | | | | | | x | | x | | |
| Generated Code Inspection | | | | x | | | | | | |
| Hardening Tools/Scripts | | | | | | | | | x | |
| Host Application Interface Scanner | | | | | | | | | x | |
| Host-Based Vulnerability Scanner | | | | | | | | | x | |
| IEEE 1028 Inspections | | | | x | | | | | | |
| Inter-Application Flow Analyzer | | | | | | | | x | x | |
| Intrusion Detection Systems/Intrusion Prevention Systems | | | | | | | | | x | |
| Logging Systems | | | | | | | | | x | |
| Man-in-the-Middle Attack Tool | | | | | | | | x | | |
| Manual Code Review | | | | x | | | | | | x |
| Mission Thread Workshop | x | | | | | | | | | |
| Negative Testing | | | | x | | x | | x | | |
| Network Sniffer | | | | | | | | | x | |
| Network Vulnerability Scanner | | | | | | | | | x | |
| Obfuscated Code Detection | | | | x | | | | | | x |
| Origin Analyzer | | | | x | x | x | x | | | |

| Tool/Technique | Stakeholder Requirements Definition | Requirements Analysis | Architectural Design | Implementation | Integration | Verification | Transition | Validation | Operation | Maintenance |
|---|---|---|---|---|---|---|---|---|---|---|
| Penetration Test | | | | | | | | x | | |
| Permission Manifest Analyzer | | | | | | | | | x | |
| Probe-Based Attack with Tracked Flow | | | | | | | | | | |
| Quality Attribute Workshop | | x | | | | | | | | |
| Rebuild and Compare | | | | x | | | | | x | |
| Safer Languages | | | x | x | | | | | | |
| Secure Library Selection | | | x | x | | | | | | |
| Secured Operating System Overview | | | | x | | | | | x | |
| Security Information and Event Management | | | | | | | | | x | |
| Software Engineering Risk Analysis | | | x | | | | | | | |
| Source Code Knowledge Extractor | | | | x | | | | | | x |
| Source Code Quality Analyzer | | | | x | | | | | | |
| Source Code Weakness Analyzer | | | | x | | | | | | x |
| Test Coverage Analyzer | | | | | | | | | | |
| Traditional Virus/Spyware Scanner | | | | x | | | | | x | |
| Web Application Vulnerability Scanner | | | | | | | | | x | |

# Appendix D: Technical Objectives

*Table 16: Technical Objectives (TO) Matrix from the SOAR Report*

*[Reprinted with permission from Wheeler and Henninger [Wheeler 2016]*

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | | | | | Situation: custom vs. OTS |
| | | | | | Data required |
| | | | | | Cost to implement |
| | | | | | SME expertise |
| | | | | | |
| **1. Provide design and code* quality** | | | | Failure to adhere to good architectural and coding standards | General: failure to adhere |
| | | | | | Use of obsolete functions |
| | | | | | Use of potentially dangerous function |
| **2. Counter known vulnerabilities (CVEs)** | | | | | |
| **3. Ensure authentication and access control*** | Authentication issues | | CWE-287 | Failure to properly authenticate users | Missing authentication for critical function |
| | | | | | Improper restriction of excessive authentication attempts |
| | | | | | Other authentication issues |

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | Credentials management | | CWE-255 | Failure to properly create, store, transmit, or protect passwords and other credentials | Use of hard-coded credentials (not put in maliciously) |
| | | | | | Other credential issues |
| | Permissions, privileges, and access control | | CWE-264 | Failure to enforce permissions or other access restrictions for resources, or a privilege management problem | Missing authorization (also, design issue) |
| | | | | | Improper/incorrect authorization |
| | | | | | Permission issues, including incorrect default permissions and incorrect permission assignment for critical resource |
| | | | | | Reliance on untrusted inputs in a security decision |
| | | | | | Other failure to enforce |
| | Least privilege* [added] | | CWE-265 | Improper enforcement of sandbox environments, or the improper handling, assignment, or management of privileges | Execution with unnecessary privileges |
| | | | | | Least privilege violation (in implementation, including grandfathering) |
| | | | | | Other privilege/sandbox issues |
| **4. Counter unintentional-"like" weaknesses** | Buffer handling* | Buffer errors | CWE-119 | Buffer overflows and other buffer boundary errors in which a program attempts to put more data in a buffer than the buffer can hold, or when a program attempts to | Incorrect calculation of buffer size |
| | | | | | Classic buffer overflow |

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | | | | put data in a memory area outside of the boundaries of the buffer | Other |
| | Injection* | Cross-site request forgery (CSRF) | CWE-352 | Failure to verify that the sender of a web request actually intended to do so. CSRF attacks can be launched by sending a formatted request to a victim, then tricking the victim into loading the request (often automatically), which makes it appear that the request came from the victim. CSRF is often associated with XSS, but it is a distinct issue. | |
| | | Cross-site scripting (XSS) | CWE-79 | Failure of a site to validate, filter, or encode user input before returning it to another user's web client | |
| | | Code injection | CWE-94 | Causing a system to read an attacker-controlled file and execute arbitrary code within that file. Includes PHP remote file inclusion, uploading of files with executable extensions, insertion of code into executable files, and others. | Unrestricted upload of file with dangerous type |
| | | | | | Download of code without integrity check |
| | | | | | Other code injection |
| | | Format string vulnerability | CWE-134 | The use of attacker-controlled input as the format string parameter in certain functions | |
| | | OS command injections | CWE-78 | Allowing user-controlled input to be injected into command lines that are created to invoke other programs, using system(s) or similar functions | |
| | | SQL injection | CWE-89 | When user input can be embedded into SQL statements without proper filtering or quoting, leading to modification of query logic or execution of SQL commands | |
| | | Input validation | CWE-20 | Failure to ensure that input contains well-formed, valid data that conforms to the application's specifications | URL redirection to untrusted site ("open redirect") [child of CWE-20] |

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | | | | Note: This overlaps other categories like XSS, numeric errors, and SQL injection. | Other input validation |
| | Encryption and randomness* | Cryptographic issues | CWE-310 | An insecure algorithm or the inappropriate use of one; an incorrect implementation of an algorithm that reduces security; the lack of encryption (plaintext); also, weak key or certificate management, key disclosure, and random number generator problems | Missing encryption of sensitive data |
| | | | | | Use of a broken or risky cryptographic algorithm |
| | | | | | Use of password hash with insufficient computational effort (incl. use of a one-way hash without a salt) |
| | | | | | Improper certificate validation |
| | | | | | Other cryptographic issues |
| | | Randomness issues | | | |
| | File handling* | Pathname traversal and equivalence errors (including link following; note that NVD uses "link following") | CWE-21 (parent of CWE-59 and CWE-22) | Failure to protect against the use of symbolic or hard links that can point to files that are not intended to be accessed by the application | Path traversal |
| | | | | | Other |
| | Information leaks* | Information leak/ disclosure | CWE-200 | Exposure of system information, sensitive or private information, fingerprinting, etc. | |
| | Number handling* | Numeric errors | CWE-189 | Integer overflow, signedness, truncation, underflow, and other errors that can occur when handling numbers | Integer overflow or wraparound |
| | | | | | Other |
| | Control flow management* | Race conditions | CWE-362 | The state of a resource can change between the time the resource is checked to when it is accessed. | |
| | | Excessive iteration | | | |

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | Initialization and shutdown [of resources/ components]* | Resource management errors | CWE-399 | The software allows attackers to consume excess resources, such as memory exhaustion from memory leaks, CPU consumption from infinite loops, disk space consumption, etc. | |
| | Design error | Design error | | A vulnerability is characterized as a “design error” if no errors exist in the implementation or configuration of a system, but the initial design causes a vulnerability to exist. [Note: Execution with unnecessary privileges moved to its own subcategory, to clearly identify it.] | Inclusion of functionality from untrusted control sphere |
| | | | | | Other design errors |
| | System element isolation | | | Design principles applied to software to allow system element functions to operate without interference from other elements | |
| | Error handling* and fault isolation | | | | |
| | Pointer and reference handling* | | | | |
| **5. Counter intentional-"like"/ malicious logic*** | Known malware | | | | Known viruses without polymorphic/metamorphic code |
| | | | | | Known viruses with polymorphic/metamorphic code |
| | | | | | Known worms |
| | | | | | Known Trojan horses (rootkits, key loggers, etc.) |
| | | | | | Other |

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | Not known malware | | | Damaging (logic) behavior not caused by common mistakes or already-known malware | Time bombs |
| | | | | | Logic bombs (condition other than time triggers failure) |
| | | | | | Back doors/trap doors (ways to get in, e.g., ports, fixed *undoc* passwords, etc.) |
| | | | | | Embedded malicious logic, e.g., Trojan horse (additional functionality not desired by user) |
| | | | | | Spyware |
| | | | | | Unrevealed "phone home" control (Note: Updates can be used this way, but are not necessarily malicious.) |
| | | | | | Application collusion (other than covert channels) |
| | | | | | Covert channel |
| | | | | | Planned/built-in obsolescence not revealed to user/acquirer |
| **6. Provide anti-tamper and ensure transparency** | Anti-tamper | Impede technology transfer (obfuscation) | | | |
| | | Impede alteration of system capability | | | |

| Technical Objective (high-level) | Technical Objective (lower-level) | Technical Objective (lower-lower-level; Source for most: NVD) | Example NVD CWE ID (where relevant) | Description (from NVD) | Technical Objective (fourth level, based on specific weaknesses) |
|---|---|---|---|---|---|
| | | Impede countermeasure development | | | |
| | Ensure transparency (detect obfuscation) † | | | | |
| **7. Counter development tool-inserted weaknesses** | Unintentional vulnerability insertion | | | | |
| | Malicious code insertion | | | | |
| **8. Provide secure delivery** | | | | | Download of code without integrity check [at delivery/installation time vs. execution time] |
| **9. Provide secure configuration** | | | CWE-16 | A general configuration problem that is not associated with passwords or permissions | |
| **10. Other** | Excessive power consumption† | | | | |

### Technical Objectives (TOs)[9]

1. Provide design and code* quality
    a. General: failure to adhere
    b. Use of obsolete functions
    c. Use of potentially dangerous function
2. Counter known vulnerabilities (CVEs)
3. Ensure authentication and access control
    a. Authentication issues
    b. Credentials management
    c. Permissions, privileges, and access control
    d. Least privilege* [added]
4. Counter unintentional-"like" weaknesses
    a. Buffer handling*
        i. Incorrect calculation of buffer size
        ii. Classic buffer overflow
        iii. Other
    b. Injection*
        i. Cross-site request forgery (CSRF)
        ii. Cross-site scripting (XSS)
        iii. Code injection
        iv. Format string vulnerability
        v. OS command injections
        vi. SQL injection
        vii. Input validation
    c. Encryption and randomness*
    d. File handling*
    e. Information leaks*
    f. Number handling*
    g. Control flow management*
    h. Initialization and shutdown [of resources/components]*
    i. Design error
    j. System element isolation
    k. Error handling* and fault isolation
    l. Pointer and reference handling*

* indicates categories that are used directly or are derived from National Security Agency recommendations

[9] A brief summary of these technical objectives is provided in the *SOAR* report.

5. Counter intentional-"like"/malicious logic*
6. Provide anti-tamper and ensure transparency
7. Counter development tool-inserted weaknesses
8. Provide secure delivery
9. Provide secure configuration
10. Other

# Appendix E: Tool Type Summary

*Table 17: Secure Development Practices from the SOAR Report*

| Secure Development Practices | Type | Grouping |
|---|---|---|
| Attack modeling | Static | Requirements |
| Warning flags | Static | Source code analyzers |
| Source code quality analyzer | Static | Source code analyzers |
| Source code weakness analyzer | Static | Source code analyzers |
| Quality analyzer | Static | Binary/bytecode |
| Bytecode weakness analyzer | Static | Binary/bytecode |
| Binary weakness analyzer | Static | Binary/bytecode |
| Inter-application flow analyzer | Static | Binary/bytecode |
| Binary/bytecode simple extractor | Static | Binary/bytecode |
| Focused manual spot check | Static | Human review |
| Manual code review | Static | Human review |
| Inspections | Static | Human review |
| Generated code inspection | Static | Human review |
| Configuration checker | Static | |
| Permission manifest analyzer | Static | |
| Host-based vulnerability scanner | Dynamic | |
| Host application interface scanner | Dynamic | |
| Web application vulnerability scanner | Dynamic | Application-type-specific vulnerability scanner |
| Web services scanner | Dynamic | Application-type-specific vulnerability scanner |
| Database scanner | Dynamic | Application-type-specific vulnerability scanner |
| Fuzz tester | Dynamic | |
| Negative testing | Dynamic | |
| Test coverage analyzer | Hybrid | |
| Hardening tools/scripts | Hybrid | |

# Appendix F: Project Context Questionnaire

Project Name _____ Date of Form _____

Project Manager Name _____

Technical Leader(s) Name _____

## I. Project Context

1. Describe the product lifecycle phase efforts for the types of work done on the project. (*Indicate percent of effort applied*)

_____ New product development

_____ Functional enhancement or upgrade of existing product

_____ Post-deployment defect fixes

_____ Migration of product or system to new platform

_____ Reengineering of existing product

_____ Other *(Please describe briefly.)*

_____

2. What portion of the physical size of the final product will use the following?

New code

Legacy code

Open source code

_____ Commercial off-the-shelf (COTS) components

_____ Government off-the-shelf (GOTS) components

3. What category best describes the project relationship with the project customer?

☐ The customer is in-house and provides project specific funding.

☐ The customer is in-house but does not directly provide project funding.

The customer is external and receives

☐ Software development services (payment for time and effort)

☐ Software for a fixed contract price

☐ Other *(Please describe briefly.)*

_____

If more than one category applies, please describe briefly.

_____

4. What category best describes the primary product use?

| | For in-house use | For external use |
|---|---|---|
| A finished product | ☐ | ☐ |
| A component or a package | ☐ | ☐ |
| A commercial product | | ☐ |

5. If more than one category applies, please describe briefly.

_____

6. Briefly describe the product or products to be produced.

_____

7. Briefly describe the customer to whom the product will be delivered.

_____

8. Briefly describe the product user (if different from the customer).

_____

9. What **single category** best describes the desired release strategy? (Please mark one box.)

☐ Product will be delivered at the end of the project.

☐ Product will be delivered incrementally during project execution, at intervals of approximately _____ weeks or _____ months.

10. What is the primary industry sector that your project supports (please consult the NAICS directory at http://www.naics.com/search.htm for additional descriptions and choices)? (Please mark one box.)

☐ Manufacturing (e.g., paper, petroleum refining, industrial and commercial machinery, computer equipment, food manufacturing, textile and apparel)

☐ Health or pharmaceutical

☐ Finance, insurance, or real estate

☐ Wholesale or retail trade

☐ Education

☐ Arts, entertainment, and recreation

☐ Telecommunications

☐ Aerospace

☐ Military (government or contractor)

☐ Other government (or government contractor)

☐ Public utilities

☐ Transportation (air, sea, or land)

☐ Other (*Please describe briefly*.)

_____

11. Which of the application domains best describes your project? (Please mark one box.)

☐ Business (e.g., decision support systems, information systems supporting business operations, payroll, accounts receivable, accounts payable, inventory)

☐ Scientific and engineering business (e.g., simulations, computer-aided design, numerical algorithms)

☐ Real-time applications (e.g., process control, manufacturing, automation, guidance systems)

☐ Embedded systems (e.g., software running in consumer electronics, keypad control for household devices, vehicles, fuel control, military systems, missile guidance)

☐ Systems software (e.g., operating systems, compilers, file management systems, editors, device drivers)

☐ Component assembly

☐ Computer-assisted software engineering (e.g., analysis and design tools, code generators, software development environments, configuration management tools, project management and cost estimation tools)

☐ Personal computer applications (e.g., word processing, spreadsheets, entertainment, games)

☐ Web applications (e.g., browsers, search engines, e-tailing, custom website development)

☐ Artificial intelligence (e.g., expert system applications, pattern recognition, learning systems)

☐ Other (*Please describe briefly.*)

_____

12. Is the system connected to other systems or networks?

_____

13. Does the system contain sensitive data that must be protected from exposure?

_____

14. Does the system contain sensitive data that must be protected from change?

_____

15. What are the entry points to interface with the system?

_____

16. How does the system provide output?

## II. Project Lifecycle Development Activities

17. What category best describes the active product lifecycle stage for the current project? (Please select only one.)

- ☐ Requirements specification (inception)
- ☐ Elaboration (early development including architecture, design, and prototyping)
- ☐ Implementation (initial development)
- ☐ Transition (a working product that may be undergoing enhancement, migration, maintenance, or sustainment)

18. What categories describe the active product development lifecycle activities for the current project? Select all that apply.

- ☐ Requirements specification
- ☐ Architectural or high-level design
- ☐ Implementation (detailed design, code, and unit test)
- ☐ Incremental implementation (may include requirements or design effort)
- ☐ Integration test (integration, functional, system, and acceptance)
- ☐ Functional or system test
- ☐ Acceptance test
- ☐ Maintenance and enhancement, including support and defect fixes
- ☐ Maintenance, including support, defect fixes only, and migration
- ☐ Maintenance including support and defect fixes only
- ☐ Other *(Please describe briefly)*

19. _____

## III. Development Environment

20. What programming language(s) do you expect to use for this project?

| **Language Used** | **Percentage Used** |
|---|---|
| _____ | _____ |
| _____ | _____ |
| _____ | _____ |

21. Will open source software be included in this product? If so, indicate the amount by language used.

| Language Used | Size |
|---|---|
| _____ | _____ |
| _____ | _____ |
| _____ | _____ |

22. Will COTS, GOTS, or other off-the-shelf binary or bytecode libraries be included in this product?

| Binary or Bytecode? | Measure (e.g., bytes/bits) | Size |
|---|---|---|
| _____ | _____ | _____ |
| _____ | _____ | _____ |
| _____ | _____ | _____ |

23. With what operating systems will the product(s) be used?

| Operating Systems |
|---|
| _____ |
| _____ |
| _____ |

24. What software development methods or tools do you expect to use in this project? (Select all that apply.)

- ☐ Formal Specification Methods (e.g., Z)
- ☐ Architecture-Centric or Architecture-Driven Development
- ☐ Rapid Prototyping (e.g., throw-away or evolutionary)
- ☐ Object-Oriented Analysis and Design)
- ☐ Modeling (e.g., UML or code generators)
- ☐ Test-Driven Development
- ☐ Automated Regression Testing
- ☐ Test Coverage Analysis
- ☐ Strategic Reuse or Architectural Product Lines
- ☐ Static Code Analysis

☐ Dynamic Code Analysis

☐ Static Architecture Analysis

☐ Information Engineering

☐ Distributed / Parallel Systems

☐ Other *(Please describe briefly.)*

_____

25. What type of revision control tools do you use on this project? (e.g., SVN, Git, ClearCase, Perforce)

☐ _____

☐ None

26. What tool or tools do you use to manage requirements?

_____

27. What tool or tools do you use to track defects in test or product lifetime?

_____

28. What other tools (e.g., IDE, modeling, code generation, and so forth) directly support development?

_____

## IV. Project Goals

29. Briefly describe the project quality goal(s).

_____

30. Rank the schedule, scope, and cost goal priorities. Please indicate the ordinal rank by goal category in order of priority (1, 2, or 3)

_____ Schedule

_____ Cost

_____ Scope or functionality

## IV.a Project Schedule

31. If in progress, when did this project begin? (Please specify month and year.)

_____ / _____

Month Year

32. When is the overall project expected to end? (Please specify month and year.)

/

Month Year

33. When is the latest acceptable end date for the overall project? (Please specify month and year.)

_____ / _____

_____ _____
Month Year

34. If part of a multi-cycle project, when is the desired end date for this cycle of the project? (Please specify month and year.)

_____ / _____
Month Year

35. If part of a multi-cycle project, what is the latest acceptable end date for this cycle? (Please specify month and year.)

_____ / _____
Month Year

36. Explain the reasons for this end date (e.g., business window, regulatory requirements).

_____

37. If this project is part of a multi-cycle project, is the overall project currently on schedule, ahead of schedule, or behind schedule?

☐ On schedule

☐ Ahead of schedule by _____ months *(Please round to nearest month.)*

☐ Behind schedule by _____ months *(Please round to nearest month.)*

## IV.b Project Cost

38. What is the expected cost for this project? Units? (e.g., dollars, pesos, euros, man days)
39. What estimation techniques support this estimate?

_____ Comparison to comparable projects

_____ Expert opinion

_____ COCOMO

_____ Slim

_____ Other(s) *(Please describe briefly.)*

40. What are planned effort requirements by specialty? (Please specify if in effort hours, days, or months.)

☐ hours ☐ days ☐ months

_____ Architects

_____ Business analysts

_____ Database designers

_____ Software designers

_____ Programmers or software engineers

_____ Testers

_____ Software quality assurance

_____ Project managers

_____ Technical writers

_____ Other(s) *(Please describe briefly.)*

_____

41. This project will likely require how many full-time and part-time staff?

_____ number of full-time staff

_____ number of part-time staff at 50% or more allocation during assignment

_____ number of part-time staff at 49% or less allocation during assignment

_____ number of contractor staff

42. If this project is part of a larger project, how many teams and sites are involved?

_____ Teams

_____ Sites

## IV.c Product Scope

43. Which method are you using to measure your product size?

- ☐ Logical SLOC
- ☐ Physical SLOC (non-commented/non-blank SLOC)
- ☐ Physical SLOC (carriage returns)
- ☐ Function Points (Which type? _____ )
- ☐ Use Cases
- ☐ Requirements Pages
- ☐ Other (e.g., story points, feature count)

  Please describe. _____

44. What is the best estimate of expected size of this product?

    with units of

45. Characterize how the development team will learn about the requirements.

- ☐ Developers use the product.
- ☐ Development team includes users of the product.
- ☐ Development team will have regular access to users of the product.
- ☐ Development team will have regular access to a user representative (e.g., analyst).
- ☐ Developers will rely on documentation for requirements.

46. Characterize the clarity and stability of the requirements. (Please select only one.)

- ☐ Requirements are documented, reviewed stable, and clearly defined.
- ☐ Requirements are documented, somewhat stable, and somewhat clear.
- ☐ Requirements are incomplete or vague.
- ☐ Requirements are highly uncertain or will change frequently.
- ☐ Unknown

47. Characterize the expected requirements change or growth. (Please select only one.)

- ☐ Requirements will change < 1% in total.
- ☐ Requirements will change < 5% in total.
- ☐ Requirements will change about 1% per calendar month.
- ☐ Requirements will change about 2% to 4% per calendar month.
- ☐ Requirements will change about 5% to 9% per calendar month.
- ☐ Requirements will change by greater than 10% per calendar month.
- ☐ Unknown

## IV.d Product Quality

48. Describe the quality goals for this product (e.g., defect density, number of user reports, mean time between failures, results from customer satisfaction surveys).

_____

49. How will quality be assured? (Select one box.)

☐ A formal QA and/or test group

☐ Development personnel with formal QA and test requirements

☐ Development personnel with informal QA and test requirements

# Appendix G: Acronyms and Abbreviations

| | |
|---|---|
| BSIMM | Building Security In Maturity Model |
| CAPEC | Common Attack Pattern Enumeration and Classification |
| CR | Code Review |
| CVE | Common Vulnerability Enumeration |
| CVSS | Common Vulnerability Scoring System |
| CWE | Common Weakness Enumeration |
| DoD | Department of Defense |
| DoDAF | Department of Defense Architectural Framework |
| DoDI | Department of Defense Instruction |
| HwA | Hardware Assurance |
| IDA | Institute for Defense Analyses |
| IDE | Integrated Development Environment |
| IEC | International Electrotechnical Commission |
| IEEE | Institute of Electrical and Electronics Engineers |
| ISO | International Organization for Standardization |
| IT | Issue Tracking |
| LOC | Lines of Code |
| MILS | Multiple Independent Levels of Separation |
| NASA | National Aeronautics and Space Administration |
| NIST | National Institute of Standards and Technology |
| NSA | National Security Agency |
| OWASP | Open Web Application Security Project |

| | |
|---|---|
| PPP | Program Protection Plan |
| RMF | Risk Management Framework |
| SA | Secure Architecture |
| SAMATE | Software Assurance Metrics and Tool Evaluation |
| SCM | Software Configuration Management |
| SD | Secure Design |
| SDL | Security Development Lifecycle |
| SDLC | Software Development Lifecycle |
| *SOAR* | State-of-the-Art Report |
| SOW | Statement of Work |
| SP | Special Publication |
| STIG | Security Technical Implementation Guide |
| SwA | Software Assurance |
| SWAPT | Software Assurance Integrated Product Team |
| TM | Threat Modeling |
| TO | Technical Objective |
| V&V | Validation and Verification |
| VCS | Version Control Software |

# Appendix H: Glossary

| | |
|---|---|
| Assurance case | A documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's properties are adequately justified for a given application in a given environment |
| Attack surface | All of the points in a system where an attacker could gain access |
| Audit | Independent review and examination of records and activities to assess the adequacy of system controls and ensure compliance with established policies and operational procedure |
| Code quality | Implies that the code implementation correctly implements the design and includes minimal defects (including known weaknesses) |
| Common Vulnerabilities Enumeration (CVE) | A list maintained by MITRE of known exploits, often mapped to weaknesses in the Common Weaknesses Enumeration (CWE) |
| Common Vulnerability Scoring System (CVSS) | An open framework for communicating the characteristics and impacts of IT vulnerabilities |
| Common Weaknesses Enumeration (CWE) | A collection by MITRE of known weaknesses that may be exploited |
| Design quality | Implies that the design fully implements the requirements without introducing unintended functional or performance problems |
| Dynamic analysis | Examines the executing software beyond traditional functional testing |
| Hybrid analysis | An analysis approach that combines elements of static and dynamic testing (e.g., test coverage analysis, hardening scripts, data tracking) |
| National Vulnerability Database (NVD) | The U.S. government repository of standards-based vulnerability management data, including databases of security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics |

| | |
|---|---|
| Open Web Application Security Project (OWASP) | An online web application security community that produces freely available articles, methodologies, documentation, and tools |
| Penetration testing | A test method in which the testers attempt to circumvent security features |
| Program Protection Plan (PPP) | The single-source document used to coordinate and integrate all protection efforts |
| Security | Involves protecting the system from unauthorized access, disclosure, or changing of data (Aspects of security include confidentiality, integrity, availability, authentication, and non-repudiation.) |
| Static analysis | Examines the code/binary/bytecode without executing it |
| System assurance (SA) | The justified confidence that the system functions as intended and the risk of vulnerabilities is managed during the software lifecycle |
| Technical objectives (TO) | Actionable approaches to addressing specific categories of design or code issues, protecting against categories of attack |
| Target of evaluation (TOE) | The artifact (typically software) that is being examined and evaluated |
| Transparency | The level to which artifacts are sufficiently understandable so that weaknesses can be discovered |
| Vulnerability | Weaknesses known to have been exploited, including but not limited to those in the Common Vulnerabilities Enumeration (CVE) |
| Weakness | A flaw in requirements, design, implementation, environment, or usage that can be exploited |

# References


### [Alberts 2003]
Alberts, Christopher & Dorofee, Audrey *Managing Information Security Risks: The OCTAVE Approach*. Addison-Wesley. 2003. https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=30678

### [Alberts 2010]
Alberts, Christopher; Allen, Julia; & Stoddard, Robert. *Integrated Measurement and Analysis Framework for Software Security*. CMU/SEI-2010-TN-025. Software Engineering Institute, Carnegie Mellon University. 2010. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9369

### [Alberts 2014]
Alberts, Christopher; Woody, Carol; & Dorofee, Audrey. *Introduction to the Security Engineering Risk Analysis (SERA) Framework.* CMU/SEI-2014-TN-025. Software Engineering Institute, Carnegie Mellon University. 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=427321

### [Austin 1996]
Austin, R. D. *Measuring and Managing Performance in Organizations*. New York, New York, USA: Dorset House. 1996.

### [Barbacci 2003]
Barbacci, Mario; Ellison, Robert; Lattanze, Anthony; Stafford, Judith; Weinstock, Charles; & Wood, William. *Quality Attribute Workshops (QAWs), Third Edition*. CMU/SEI-2003-TR-016. Software Engineering Institute, Carnegie Mellon University. 2003. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6687

### [Barnes 2009]
Barnes, J. *The Spark Approach to High Integrity Software.* John Barnes Infomatics and Praxis Critical Systems. June 8, 2009. https://www.ada-deutschland.de/sites/default/files/tagungen/ws2009/presentations/001_Barnes_Regensburg.pdf

### [Bartock 2016]
Bartock, M.; Cichonski, J.; Souppaya, M.; Smith, M.; Witte, G.; & Scarfone, K. *Guide for Cybersecurity Event Recovery*. NIST Special Publication 800-184. https://doi.org/10.6028/NIST.SP.800-184

### [Basili 1992]
Basili, V. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. University of Maryland. CS-TR-2956, UMIACS-TR-92-96. September 1992. https://www.cs.umd.edu/~basili/publications/technical/T78.pdf

**[Black 2008]**
Black, P. E.; Scarfone, K.; & Souppaya, M. Cybersecurity Metrics and Measures. *Wiley Handbook of Science and Technology for Homeland Security.* 2008. https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=51292

**[Boyens 2015]**
Boyens, J.; Paulsen, C.; Moorthy, R.; & Bartol, N. *Supply Chain Risk Management Practices for Federal Information Systems and Organizations*. 2015. https://doi.org/10.6028/NIST.SP.800-161

**[Butler 2016]**
Butler, Greg; Kendall, Kim; & Hunsinger, Stephani. *Cybersecurity in the Systems Engineering Process: A High Level SE Process Orientation*. Defense Acquisition University. June 2016.

**[Chillarege 1992]**
Chillarege, R.; Bhandari, I. S.; Chaar, J. K.; Halliday, M. J.; Moebus, D. S.; Ray, B. K.; & Wong, M.-Y. Orthogonal Defect Classification – A Concept for In-process Measurements. *IEEE Transactions on Software Engineering*. Volume 18. Issue 11. Pages 943–956. 1992. https://doi.org/10.1109/32.177364

**[Clark 2015]**
Clark, T. Most Cyber Attacks Occur From This Common Vulnerability. *Forbes.* March 10, 2015. https://www.forbes.com/sites/sap/2015/03/10/most-cyber-attacks-occur-from-this-common-vulnerability/#415f45a77454

**[CMMI 2010]**
CMMI Product Team. *CMMI for Acquisition, Version 1.3.* CMU/SEI-2010-TR-032. Software Engineering Institute, Carnegie Mellon University. 2010. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9657

**[DoD 2011]**
United States Department of Defense. *Program Protection Plan Outline & Guidance Version 1.0*. 2011. https://www.acq.osd.mil/se/docs/PPP-Outline-and-Guidance-v1-July2011.pdf

**[Ellison 2015]**
Ellison, Robert; Householder, Allen; Hudak, John; Kazman, Rick; & Woody, Carol. *Extending AADL for Security Design Assurance of Cyber-Physical Systems*. CMU/SEI-2015-TR-014. Software Engineering Institute, Carnegie Mellon University. 2015. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=449510

**[Feiman 2014]**
Feiman, J. Maverick Research: Stop Protecting Your Apps; It's Time for Apps to Protect Themselves. G00269825. *Gartner*. 2014. www.gartner.com/doc/2856020/maverick-research-stop-protecting-apps

**[Gagliardi 2013]**
Gagliardi, Michael; Wood, William; & Morrow, Timothy. *Introduction to the Mission Thread Workshop*. CMU/SEI-2013-TR-003. Software Engineering Institute, Carnegie Mellon University. 2013. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=63148

**[Goethert 2004]**
Goethert, Wolfhart & Siviy, Jeannine. *Applications of the Indicator Template for Measurement and Analysis.* CMU/SEI-2004-TN-024. Software Engineering Institute, Carnegie Mellon University. 2004. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6969

**[Goodenough 2012]**
Goodenough, John; Weinstock, Charles; & Klein, Ari. *Toward a Theory of Assurance Case Confidence.* CMU/SEI-2012-TR-002. Software Engineering Institute, Carnegie Mellon University. 2012. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28067

**[Heffley 2004]**
Heffley, J. & Meunier, P. Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security? *Proceedings of the 37th Hawaii International Conference on System Sciences*. Institute of Electrical and Electronics Engineers. 2004. https://doi.org/10.1109/HICSS.2004.1265654

**[Humphrey 1995]**
Humphrey, W. S. A Discipline for Software Engineering. *Lecture Notes in Computer Science.* Volume 640. Reading, MA: Addison-Wesley Longman Publishing Co. 1995. http://www.amazon.com/Discipline-Software-Engineering-Watts-Humphrey/dp/0201546108

**[ISO/IEC 2008]**
ISO/IEC 12207:2008. Systems and Software Engineering -- Software Lifecycle Processes. 2008.

**[ISO/IEC 2011]**
ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQUARE)—System and Software Quality Models. 2011.

**[ISO/IEC 2013]**
ISO/IEC 15026-1:2013. Systems and Software Engineering—Systems and Software Assurance—Part 1: Concepts and Vocabulary. 2013.

**[ISO/IEC/IEEE, 2015]**
ISO/IEC/IEEE 15288:2015. Systems and Software Engineering—System Lifecycle Processes. 2015.

**[Jones 2009]**
Jones, C. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. New York: McGraw-Hill Osborne Media. 2009.

**[Jones 2011]**
Jones, C. & Bonsignour, O. *The Economics of Software Quality.* Addison-Wesley Professional. 2011.

**[Kan 2002]**
Kan, S. H. Software Quality Metrics Overview. *In Metrics and Models in Software Quality Engineering*. Pages 85–126. Addison-Wesley Professional. 2002.

**[Kazman 2000]**
Kazman, Rick; Klein, Mark; & Clements, Paul. *ATAM: Method for Architecture Evaluation.* CMU/SEI-2000-TR-004. Software Engineering Institute, Carnegie Mellon University. 2000. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5177

**[Lautieri 2005]**
Lautieri, S.; Cooper, D.; & Jackson, D. SafSec: Commonalities Between Safety and Security Assurance. *Constituents of Modern System-Safety Thinking - Proceedings of the 13th Safety-Critical Systems Symposium*, February 2005. Pages 65–75. https://doi.org/10.1007/1-84628-130-X-5

**[Leveson 2004]**
Leveson, N. A New Accident Model for Engineering Safer Systems. *Safety Science*. Volume 42. Issue 4. 2004. Pages 237–270. https://doi.org/10.1016/S0925-7535(03)00047-X

**[Mani 2014]**
Mani, Sidhartha & Mead, Nancy. *An Evaluation of A-SQUARE for COTS Acquisition.* CMU/SEI-2014-TN-003. Software Engineering Institute, Carnegie Mellon University. 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=90536

**[McDermott 1999]**
McDermott, J. & Fox, C. Using Abuse Case Models for Security Requirements Analysis. *Proceedings 15th Annual Computer Security Applications Conference*. IEEE Computer Society. 1999. Pages 55–64. https://doi.org/10.1109/CSAC.1999.816013

**[McGraw 2017]**
McGraw, G.; Migues, S.; & West, J. *Building Security In Maturity Model Version 8*. 2017. https://www.bsimm.com/content/dam/bsimm/reports/bsimm8.pdf

**[Mead 2016]**
Mead, Nancy R. & Woody, Carol C. *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance.* Addison-Wesley Professional. 2016.

**[Microsoft 2002]**
Microsoft. *The STRIDE Threat Model*. 2002. https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx

**[MITRE 2017]**
MITRE. *About CWE: Frequently Asked Questions*. January 2017. https://cwe.mitre.org/about/faq.html#A.2

**[MITRE 2018a]**
MITRE. *Common Attack Pattern Enumeration and Classification—CAPEC, A Community Knowledge Resource for Building Secure Software*. 2018. http://makingsecuritymeasurable.mitre.org/docs/capec-intro-handout.pdf

**[MITRE 2018b]**
MITRE. *The Common Weakness Enumeration (CWETM) Initiative.* 2018. https://cwe.mitre.org/

**[NDIA 2008]**
NDIA System Assurance Committee. *Engineering for System Assurance.* National Defense Industrial Association. Arlington, VA. 2008. http://www.acq.osd.mil/se/docs/SA-Guidebook-v1-Oct2008.pdf

**[Nichols 2012]**
Nichols, W. R. Plan for Success, Model the Cost of Quality. *Software Quality Professional*. Volume 14. Issue 2. Pages 4–11. 2012.

**[Nichols 2018]**
Nichols, William; McHale, James; Sweeney, David; Snavely, William; & Volkmann, Aaron. *Composing Effective Software Security Assurance Workflows.* CMU/SEI-2018-TR-004. Software Engineering Institute, Carnegie Mellon University. 2018. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=528467

**[NIST 2010]**
NIST. *Guide for Applying the Risk Management Framework to Federal Information Systems.* NIST Special Publication 800-37, Rev 1February, 2010. https://doi.org/NIST Special Publication 800-37 R1

**[NIST 2014]**
NIST. *Security and Privacy Controls for Federal Information Systems and Organizations Security and Privacy Controls for Federal Information Systems and Organizations.* Sp-800-53Ar4, 400+. 2014. https://doi.org/10.6028/NIST.SP.800-53Ar4

**[Park 1996]**
Park, R. E.; Goethert, W. B.; & Florac, W. A. *Goal-Driven Software Measurement—A Guidebook*. CMU/SEI-96-HB-002. Software Engineering Institute, Carnegie Mellon University. 1996. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=12453

**[Paulsen 2018]**
Paulsen, C.; Boyens, J.; Bartol, N.; & Winkler, K. *Criticality Analysis Process Model: Prioritizing Systems and Components.* NISTIR 8179. 2018. https://doi.org/10.6028/NIST.IR.8179

**[Petersen 2009]**
Petersen, K. & Wohlin, C. Context in Industrial Software Engineering Research. *3rd International Symposium on Empirical Software Engineering and Measurement.* Pages 401–404. 2009. https://doi.org/10.1109/ESEM.2009.5316010

**[RTCA 2012]**
RTCA DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. 2012.

**[Salter 1998]**
Salter, C.; Saydjari, O.; Schneier, B.; & Wallner, J. Toward a Secure System Engineering Methodology. *New Security Paradigms Workshop*. Pages 2-10. September 1998. https://www.schneier.com/academic/archives/1998/09/toward_a_secure_syst.html

**[Saltzer 1974]**
Saltzer, J. H. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, Volume 17. Issue 7. 1974. Pages 388-402. https://doi.org/10.1145/361011.361067

**[Saltzer 1975]**
Saltzer, J. H. & Schroeder, M. D. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, Volume 63. Issue 9. 1975. Pages 1278–1308. https://doi.org/10.1109/PROC.1975.9939

**[Sindre 2005]**
Sindre, G. & Opdahl, A. L. Eliciting Security Requirements with Misuse Cases. *Requirements Engineering*, Volume 10. Issue 1. 2005. Pages 34–44. https://doi.org/10.1007/s00766-004-0194-4

**[Shevchenko 2018a]**
Shevchenko, N.; Chick, T. A.; Riordan, P. O.; Scanlon, T. P.; & Woody, C. *Threat Modeling: A Summary of Available Methods*. 2018. Pittsburgh, PA. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=524448

**[Shevchenko 2018b]**
Shevchenko, N.; Frye, B. R.; & Woody, C. *Threat Modeling for Cyber-Physical System-of-Systems: Methods Evaluation*. 2018. Pittsburgh, PA. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=526365

**[Swiderski 2009]**
Swiderski, F. & Snyder, W. *Threat Modeling.* O'Reilly Media, Inc. 2009.

**[USD(AT&L) 2017]**
USD(AT&L). *Department of Defense Instruction 5000.02.* Department of Defense. 2017. http://acqnotes.com/wp-content/uploads/2014/09/DoD-Instruction-5000.02-The-Defense-Acquisition-System-10-Aug-17-Change-3.pdf

**[Viega 2002]**
Viega, J. & McGraw, G. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley. 2002.

**[Wheeler 2016]**
Wheeler, D. A. & Henninger, A. *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation V2.2*. 2016. https://www.acq.osd.mil/se/docs/P-8005-SOAR-2016.pdf

**[Woody 2014]**
Woody, Carol; Ellison, Robert; & Nichols, William. *Predicting Software Assurance Using Quality and Reliability Measures.* CMU/SEI-2014-TN-026. Software Engineering Institute, Carnegie Mellon University. 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=428589

**[Woody 2015]**
Woody, Carol; Ellison, Robert; & Nichols, William. Predicting Cybersecurity Using Quality Data. In *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*. Pages 1-5. 2015. Waltham, MA: IEEE. https://doi.org/10.1109/THS.2015.7225327

**[Woody 2016]**
Woody, Carol. Security Engineering Risk Analysis (SERA). *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice*. Pages 23–24. New York, New York, USA: ACM Press. 2016. https://doi.org/10.1145/2897022.2897024

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| (Leave Blank) | December 2018 | Final |

| 4. TITLE AND SUBTITLE | 5. FUNDING NUMBERS |
|---|---|
| DoD Developer's Guidebook for Software Assurance | FA8702-15-D-0002 |

6. AUTHOR(S)

Dr. William R. Nichols, Jr. and Dr. Thomas Scanlon

| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) | 8. PERFORMING ORGANIZATION REPORT NUMBER |
|---|---|
| Software Engineering Institute<br>Carnegie Mellon University<br>Pittsburgh, PA 15213 | CMU/SEI-2018-SR-013 |

| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER |
|---|---|
| SEI Administrative Agent<br>AFLCMC/AZS<br>Enterprise Acquisition Division<br>5 Eglin Street<br>Hanscom AFB, MA 01731-2100 | n/a |

11. SUPPLEMENTARY NOTES

| 12A DISTRIBUTION/AVAILABILITY STATEMENT | 12B DISTRIBUTION CODE |
|---|---|
| Unclassified/Unlimited, DTIC, NTIS | |

13. ABSTRACT (MAXIMUM 200 WORDS)

Software assurance refers to the justified confidence that software functions as intended and is free of vulnerabilities throughout the product lifecycle. While "free of vulnerabilities" is the ideal, in practice the objective is to manage the risk associated with vulnerabilities. To that end, this guidebook helps software developers understand expectations for software assurance. Because developers need to be aware of the regulatory background in which their projects operate, this guidebook summarizes standards and requirements that affect software assurance decisions and provides pointers to key resources that developers should consult. It includes a summary of the State-of-the-Art Resources (*SOAR*) for Software Vulnerability Detection, Test, and Evaluation report, along with its approach for selecting tools. A bottom-up approach to tool selection is also provided, which considers what activities and tools are typically appropriate at different stages of the development or product lifecycle. Advice is provided for special lifecycle considerations, such as new development and system reengineering, and metrics that may be useful in selecting and applying tools or techniques during development are discussed. Special sections are devoted to assurance in software sustainment and software acquisition. Supplemental materials are provided in the appendices.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| software assurance, cybersecurity, development, security tools, standards, requirements, measurement | 111 |

16. PRICE CODE

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102