

Code Risk Estimation Worksheet (CREW) v5.3.0 User Manual

Alan Cohn
Ryan Karl
Mena Kostial
Jay Marchetti
Nicholas Reimer
Michael Riley
David Svoboda

May 2023

[Distribution Statement A] Approved for public release and unlimited distribution.

<http://www.sei.cmu.edu>



Copyright 2023 Carnegie Mellon University.

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.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material is distributed by the Software Engineering Institute (SEI) only to course attendees for their own individual study.

Except for any U.S. government purposes described herein, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at permission@sei.cmu.edu.

Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM23-0505

Table of Contents

Abstract	iv
1 Introduction	1
1.1 Code Risk Estimation Worksheet (CREW) Overview	1
1.2 Background Information	1
1.3 How To Use This Document	2
1.4 Getting Started	2
2 Risk Results	3
2.1 Overview	3
3 Inputs	5
3.1 Overview	5
4 System	6
4.1 Overview	6
4.2 Basis	6
4.3 Metrics	7
5 Architecture	9
5.1 Overview	9
5.2 Getting Started	10
5.3 Modularity	10
5.4 Dependency	14
5.5 Patterns	15
6 Standards	19
6.1 Overview	19
6.2 Getting Started	20
6.3 Stylistic	21
6.4 Criticality	26
6.5 Language C/C++	33
6.6 Language C++	43
7 Maintainability Comments	58
7.1 Overview	58
7.2 Quantity	59
7.3 Quality	62
7.4 Script Outputs	72
7.5 Various Scoring Examples	76
8 Maintainability Clarity	78
8.1 Overview	78
8.2 Consistency	79
8.3 Constants	85
8.4 Files / Folders	90
8.5 Readability	94
8.6 Conditionals	101
8.7 Script Outputs	102
9 Maintainability Reusability	104

9.1	Overview	104
9.2	Independence	105
9.3	Utilization	113
9.4	Complication	115
9.5	Duplication	118
9.6	Script Outputs/Examples	121
10	Fault Tolerance	126
10.1	Overview	126
10.2	Value Checking	127
10.3	Error Handling	132
10.4	Error Logging	134
10.5	Instrumentation	136
10.6	Health Monitoring	140
11	Testability	142
11.1	Overview	142
11.2	Unit	143
11.3	Regression	147
11.4	Complexity	149
12	Security General	152
12.1	Overview	152
12.2	Defensiveness	153
12.3	Validation	158
12.4	Libraries	160
12.5	Warnings	162
12.6	Traceability	163
13	Security CERT	164
13.1	Overview	164
14	CERT Summary	178
14.1	Overview	178
14.2	SCALE Introduction	178
14.3	SCALE User Roles	181
14.4	SCALE Audit Basic Instructions	182
14.5	The SCALE Web App	182
14.5.1	Uploading alerts from Static Analyzers	183
14.5.2	Inspecting Alerts for Violations of CERT Secure Coding Rules	187
14.5.3	The Alert Viewer	188
14.5.4	The Auditing Workflow	194
15	Tools	210
15.1	Overview	210
15.2	Tools	210
15.3	Analytic Scripts	211
16	Conclusion	214
17	CREW Manual Change Log	215
	References/Bibliography	216

Abstract

Recent high-profile cyberattacks on critical US infrastructure highlight the risks poor code quality poses to the American economy and national security. To help identify codebases that pose significant risk to American assets, the Code Risk Estimation Worksheet (CREW) was developed by SEI as a framework for conducting independent code analyses of government computing systems. CREW assessments help identify the *structural code quality*, e.g., the quality of the code itself, rather than *functional code quality*, e.g., how well the code fulfills mission requirements.

CREW assists technology professionals with analyzing code quality within the context of risk management, by providing a statistical assessment of the risks associated with various attributes of a codebase. Attributes identified as having high risk scores can then be targeted for actionable improvements or mitigations.

This document serves as a user manual for CREW analysts and provides guidance on what methodologies to use when answering questions contained in CREW v5.3.0. After providing a brief overview of the tools, techniques, and procedures utilized in CREW assessments, instructions are provided on how to answer each individual question contained in each tab of the CREW spreadsheet.

1 Introduction

1.1 Code Risk Estimation Worksheet (CREW) Overview

The Code Risk Estimation Worksheet (CREW) is a framework developed by the SEI to assist with assessing the risk of poor code quality in embedded systems. CREW attempts to objectify risk estimation by directing analysts to answer code-level questions after leveraging static analysis (SA) tools and reading subsections of the source code [Marchetti 2021]. CREW is implemented as an Excel worksheet and is comprised of questions that guide analysts through the SEI's code risk estimation process.

CREW investigates six (6) aspects of structural code quality to assess the overall level of risk: implemented architecture, standards adherence, maintainability, fault tolerance, security, and testability. Each code quality aspect is further subdivided into two (2) or more foci. Answers to each question are aggregated and a risk score in the closed interval [1.0, 5.0] is calculated for each focus. These scores are combined to generate a risk score for each of the six aspects, which is used to assign a total risk estimate for the entire codebase. If a customer is interested in additional aspects or would like to exclude certain aspects from the analysis, CREW can be modified to meet this need.

CREW assessments are typically performed on classified or unclassified C/C++ code specific to mission-, safety-, or security-critical embedded systems, where access to design documentation or original developers is not possible. For more information see the SEI's whitepaper "Assessing Software Quality Using a Risk-Based Methodology" [Marchetti 2021].

1.2 Background Information

To successfully complete a CREW assessment, some background knowledge regarding the traditional code audit process is required. A brief review of the tools, techniques, and procedures commonly used during such audits is provided in the following paragraphs.

Static Analysis

Static Analysis (SA) is the analysis of software while it is not executing [Dewhurst 2022]. SA may be performed on a variety of code languages, including C/C++ or Java source code, Java or .NET bytecode, or on binaries, to diagnose security concerns about the quality of the software. Modern SA best practices typically analyze source code by leveraging automated or semi-automated analysis tools. SA is commonly instrumented by using multiple SA tools in tandem and should be done throughout the software development process [SEI 2022].

SA tools typically have the capability to scan lines of code and report which lines violate coding standard rules, such as ISO/IEC 25010, software safety standards such as DO-178C for avionics systems, and software security standards such the SEI CERT Secure Coding Standards for C, C++, and Java. Leveraging this capability is often an important component of a code audit, but it should be noted that SA tools sometimes report false positive violations. Human-in-the-loop adjudication should be used to ascertain the validity of any violations reported by an SA tool. SEI

does not endorse any vendor or tool, but a list of tools typically used when SEI performs a CREW analysis can be found in ‘Tools’ section of this document.

Regular Expressions

A regular expression, sometimes referred to as a regex, is a powerful tool for manipulating text and data [Friedl 2006]. Regexes are often used by analysts performing a code audit to search for keywords or text string patterns within a codebase that are indicative of problematic code. Many CREW questions utilize regexes to search for high-risk code by using the built-in regex search capabilities provided by most SA tools. A comprehensive discussion of regexes is beyond the scope of this document, but additional information can be found in external resources [Friedl 2006].

Sampling Procedures

When performing a code audit, it may not be feasible to manually inspect every file in a codebase. In such circumstances, analysts may choose to review a representative sampling of all the files in the codebase. It is recommended that each CREW analyst review ~10% of the files in the codebase or ~100-200 files, whichever is less. Sampled files should generally be distributed across the entire codebase, however, it is recommended that analysts also include files with high cyclomatic complexity, high SLOC, or other peculiar characteristics within their analysis, as these files more often contain problematic code.

1.3 How To Use This Document

This document serves as a user manual for CREW analysts and provides guidance on what methodologies to use when answering questions contained in CREW v5.3.0. The following chapters provide an overview of each CREW tab, followed by instructions on how to answer each individual question contained in the given tab.

1.4 Getting Started

CREW analysis teams usually designate a team member to be the build master. The build master is responsible for building the software from source to verify the source code is complete, and generating the preliminary SA tool output (e.g., Understand project) used during the CREW assessment. In practice, several SA tool rule scans and regex searches may be computed in advance of performing the CREW assessment, to speed up the overall CREW process and reduce the amount of duplicate work.

2 Risk Results

2.1 Overview

The Risk Results tab is CREW's top-level output view for code quality assessments. The Risk Results tab presents a synopsis of the code analysis which shows basic information regarding the codebase such as the system name, revision, language, and run-time environment, as input on the System tab. The Risk Results tab then displays the rolled-up risk scores for all 32 foci, the 6 quality aspects, and an overall risk score for the codebase. The overall risk score derives from the weighted scores across all the analyst tabs. The color-coded score summary view is central in all SEI CREW code quality out briefs. The Risk Results tab is shown in Figure 2.1.1:

Figure 2.1.1 The Risk Results Tab

Code Risk Estimation Worksheet									
CREW v5.24 Software Engineering Institute - ©2023 Carnegie Mellon University				USAGE KEY					
Project: SystemX Code Revision: 1.52 Language(s): C Run-time OS: VxWorks 6.9 Run-time CPU: Unknown Analyst(s): Review Date: 08/16/22				Aspect: The six (6) aspects of code quality being assessed for risk. Focus: Sub-aspects that are rated via the nine (9) scoring tabs. Best Practice: Focus best practice / positive code quality characteristic. Change: Optional analyst's noted change in a rating from the assessment of a prior code drop (input [+/- delta]). Analyst: Optional assigned analyst for an aspect (input [text]). Notes: Optional additional analyst's notes (input [text]). Ratings: Rolled-up risks from the nine (9) scoring tabs that the best practice statements are NOT true. 1.0 2.0 3.0 4.0 5.0 (Low Risk) (High Risk)					
CODE RISK ESTIMATION WORKSHEET FOR: SystemX									
ASPECT RISK	ASPECT / Focus	BEST PRACTICE		CHANGE	ANALYST	NOTES			
2.4	ARCHITECTURE /								
	2.1 Modularity	High level of code modularity and module cohesiveness							
	2.4 Dependency	Low level of dependencies, especially cyclic (DSM, dependency graphs, etc)							
	2.7 Patterns	High level of adherence to a known architectural pattern (layered, client-server, etc)							
3.4	STANDARDS /								
	2.5 Stylistic	High level of adherence to a chosen internal or published stylistic coding standard							
	3.1 Criticality	High level of adherence to critical-system coding guidelines (MISRA, CERT, etc)							
	3.9 Language	High level of adherence to widely accepted language guidelines							
2.8	MAINTAINABILITY /								
Comments	1.7 Quantity	High level of commenting excluding boilerplates							
	2.2 Quality	High quality of commenting that clarifies the code's intent							
Clarity	1.4 Consistency	High level of code stylistic consistency (naming, source layout, brace usage, etc)							
	2.7 Constants	High level of use of symbolic constants							
	1.6 Files / Folders	High level of source file and folder naming consistency and organization							
	3.2 Readability	High level of overall code readability							
	4.0 Conditionals	Low level of conditional compilation							
Reusability	2.3 Independence	High level of machine, OS, database, and compiler independence							
	3.4 Utilization	Low level of commented-out or dead code, and unused variables							
	4.0 Complication	Low level of complicated implementation constructs							
	5.0 Duplication	Low level of duplicated (copy / pasted) code							
3.8	FAULT TOLERANCE /								
	3.8 Value Checking	High level of return and parameter value checking							
	4.7 Error Handling	Existence of a systematic error handling mechanism							
	3.0 Error Logging	Existence of an error / fault logging mechanism							
	3.8 Instrumentation	Existence of run-time statistics and timing instrumentation							
	2.9 Health Monitoring	Effective use of watchdog and system health monitoring							
3.2	SECURITY /								
General	3.6 Defensiveness	High level of defensive / secure coding practices							
	3.8 Validation	High level of validation of untrusted inputs / interfaces							
	3.0 Libraries	Low level of use of potentially unsafe library calls							
	2.3 Warnings	Evidence that code compiles with no (or few) warnings							
	3.9 Traceability	Effective use of executable and parameter file signatures / anti-tamper measures							
CERT	2.9 Density	Low density of secure coding issues as per SEI CERT Secure Coding Rules							
	4.0 Priority	Low number of high-priority secure coding issues as per SEI CERT Secure Coding Rules							
3.4	TESTABILITY /								
	2.6 Unit	Evidence of unit testing, especially automated (test cases, test harness, etc)							
	4.0 Regression	Evidence of regression testing, especially automated							
	3.9 Complexity	Low valued cyclomatic complexity and path count							
	3.17	TOTAL RISK							

While the Risk Results tab is primarily a display (output) tab, several pieces of information may be entered on it:

- The grey box at the top may be edited to indicate the surname(s) of the individual or team who performed the analysis. It also contains the analysis completion date.
- The Change field can be used to enter a $\pm$ delta from a prior CREW analysis of a code-base when analyzing multiple code drops. This value is simply the current (this) focus score minus the prior CREW analysis focus score, such that a negative delta indicates improvement for that focus.
- The Analyst field can be used to identify analysts. This may be useful to parcel out and record the surname of who on the analysis team performed the analysis for each aspect or focus.
- The Notes field, while seldom used by SEI, may be utilized to highlight or annotate a particular aspect or focus if desired.

3 Inputs

3.1 Overview

This tab tracks which CREW inputs have been received for the code analysis. A CREW analyst will type “Yes” or “No” in the column labeled “Received” to indicate the status of the artifacts.

It is preferred that inputs marked as “High” in the “Priority” column are received before the analysis commences, though quite often this is not the case.

This tab also serves as an archival record of what artifacts were provided for the completed CREW analysis. The Inputs tab is shown in Figure 3.1.1.

Figure 3.1.1 The Inputs Tab

#	Priority	Received	Artifact	Note
1	High	Yes	All source code needed to build the system organized in the proper subfolder tree for the build.	
2	High	Yes	All build scripts, project files, solution files, or makefiles needed to build the system executable(s) from the supplied source code files.	
3	High	No	Specification: Build Instructions	
4	High	Yes	A complete build log showing the build results from / verifying the build.	
5	High	Yes	A complete list, including precise version numbers, for all compilers, linkers, target RTOS's / OS's, build host OS's, build host IDE's, preprocessing tools, loaders / debuggers, and any other utilities used to build the system executable(s).	
6	High	Yes	All unit test code, mocks, and supporting unit test framework code.	
7	High	No	The precise ID (i.e. manufacturer's part number) of the processor and board (if COTS) for which the executable(s) are targeted.	
8	High	No	A complete list, including precise version numbers, for all code generation tools utilized.	
9	High	No	A virtual machine (VM) .OVF file containing the entire source, build system, and all required tools that demonstrates / verifies the build.	
10	Medium	Yes	Specification: Concept of Operations (CONOPS)	
11	Medium	No	Specification: Software Development Plan (SDP)	
12	Medium	No	Specification: System Architecture Specification (SAS)	
13	Medium	Yes	Specification: Software Design Specification(s) (SDS)	
14	Medium	No	Specification: Software Test Specification (STS)	
15	Medium	No	Specification: Software Deployment Plan for executable(s) installation & upgrades	
16	Medium	No	Specification: Coding Standard (custom or adopted)	
17	Low	No	Other relevant process documents utilized for the system including: References for critical system coding standards used, a list of any static analysis tools employed, a list of any version control tool(s) and documentation of the branching strategy used, a list of any defect tracking tool(s) used.	
18	Low	No	Files identifying the current prioritized Problem Report (PR) backlog and the six (6) most recent months of PR burn-down trend data.	
19	Low	No	Any linkable library files needed (board support, RTOS, peripherals, etc.) to build the system executable(s) from the supplied source code files.	
20	Low	No	The system executable(s) built from the supplied source code.	

4 System

4.1 Overview

This tab is used to collect basic information about the codebase, and statistical data regarding the size and structure of the codebase, such as the total number of files, lines of code, etc. Much of this data is typically metrics from SA tools that is utilized within other tabs (See questions where the field labeled “A” for Automatic contains a ● dot). However, these metrics are also useful for gaining a high-level understanding of the codebase. Note that the System tab does not directly affect risk scores, though the metrics it provides for use in other tabs are utilized within scored questions on those (client) tabs. Because other tabs rely on this data, it is recommended that this tab be completed first. Note that alternate versions of the same SA tool sometimes compute different statistics for the same codebase. The System tab is shown in Figure 4.1.1.

Figure 4.1.1 The System Tab

Date:	11/22/22			
Analyst:	Marchetti			
SA Tool Used:	Understand 6.2 (Build 1125)			
System	ID	Measurement / Question	Finding	Note / Guidance
BASIS	SY__BA01	System Name	WMI22	0
	SY__BA02	Build environment OS and version	Ubuntu 18.04	0
	SY__BA03	Compiler and version	gcc 7.5.0	0
	SY__BA04	Source code language	C++	0
	SY__BA05	Source code / system revision if known	1.3.0	0
	SY__BA06	Run-time OS and version if known	Unknown	0
	SY__BA07	Run-time CPU if known	Unknown	0
METRICS	SY__ME01	Total number of source files	1882	0
	SY__ME02	Number of definition (header) files	945	0
	SY__ME03	Number of implementation (code) files	937	0
	SY__ME04	Total number of (all) lines in the code base	631056	0
	SY__ME05	Source lines of code ignoring blank and comment lines (PSLOC)	382397	0
	SY__ME06	Executable source lines of code (ESLOC)	38640	0
	SY__ME07	Number of lines in the code base containing a comment (CLOC)	147616	0
	SY__ME08	Ratio of number of comment lines to number of code lines	39%	0
	SY__ME09	Total number of functions or methods within the code base	8157	0
	SY__ME10	Average file cyclomatic complexity (CC1)	2.58	0
	SY__ME11	Maximum cyclomatic complexity (CC1) for any function within the code base	151	0
	SY__ME12	Maximum control structure nesting level within the code base	10	0
	SY__ME13	Cyclomatic complexity (total CC1) of the entire code base v(G)	17907	0
	SY__ME14	Essential complexity of the entire code base Ev(G)	9097	0
	SY__ME15	Estimated total number of commented-out lines of code	1035	0

The following sections describe how to score the CREW System tab:

4.2 Basis

This section captures the basis information for the codebase to be analyzed.

SY__BA01: System Name

Enter the friendly name of the system, as per the developing organization or customer. If the code to be analyzed is a subset of a larger codebase, consider a name that identifies the subset.

SY__BA02: Build environment OS and version

Obtain this information from the analyst responsible for doing the build. SEI designates one analyst, the team’s build master, for the task of achieving a complete compilation of the system to be analyzed. This is required to obtain accurate output from most static analysis tools.

SY__BA03: Compiler and version

Obtain this information from the analyst responsible for doing the build (the team's build master).

SY__BA04: Source code language

Scan the code to determine if it is pure C, pure C++, or a mixture of both. This may include reviewing source file extensions, reviewing the supplied Makefiles, searching for C++ specific constructs within the codebase (e.g., "class"), and discussion with the build master.

SY__BA05: Source code / system revision if known

Enter the version identification of the system, as per the developing organization or customer, if known.

SY__BA06: Run-time OS and version if known

Enter the run-time OS and its version for the system, as per the developing organization or customer, if known.

SY__BA07: Run-time CPU if known

Enter the run-time CPU and / or target computing hardware, as per the developing organization or customer, if known.

4.3 Metrics

This section captures the top-level metrics that are utilized within other CREW tabs. Use a SA tool (e.g. Parasoft) to determine these metrics. Each metric's API or CodeCheck rule for the SA tool Understand (Scientific Toolworks) is included below in parentheses.

SY__ME01: Total number of source files

Enter the total number of C/C++ source code files (CountDeclFile).

SY__ME02: Number of definition (header) files

Enter the total number of C/C++ header (.h, .hpp, etc.) files (CountDeclFileHeader).

SY__ME03: Number of implementation (code) files

Enter the total number of C/C++ implementation (.c, .cpp, .C, etc.) files (CountDeclFileCode).

SY__ME04: Total number of (all) lines in the code base

Enter the total number of source lines (CountLine aka NL).

SY__ME05: Source lines of code ignoring blank and comment lines (PSLOC)

Enter the total number of source lines excluding blank and comment-only lines, e.g. Physical SLOC (CountLineCode).

SY_ME06: Executable source lines of code (ESLOC)

Enter the total number of executable lines of code (CountLineCodeExe).

SY_ME07: Number of lines in the code base containing a comment (CLOC)

Enter the total number of comment(ed) lines (CountLineComment).

SY_ME08: Ratio of number of comment lines to number of code lines

Enter the ratio of comment(ed) lines to Physical SLOC. Divide the above values or obtain from the SA tool (RatioCommentToCode).

SY_ME09: Total number of functions or methods within the code base

Enter the total number of functions within the code (CountDeclFunction).

SY_ME10: Average file cyclomatic complexity (CC1)

Enter the average file cyclomatic complexity for the code (AvgCyclomatic).

SY_ME11: Maximum cyclomatic complexity (CC1) for any function within the code base

Enter the maximum function cyclomatic complexity within the code (MaxCyclomatic).

SY_ME12: Maximum control structure nesting level within the code base

Enter the maximum depth level of source code control structure nesting (e.g., if(if()) would be 2) in the code (MaxNesting).

SY_ME13: Cyclomatic complexity (total CC1) of the entire code base $v(G)$

Enter the summation of cyclomatic complexity for the entire code (Cyclomatic).

SY_ME14: Essential complexity of the entire code base $Ev(G)$

Enter the summation of essential complexity for the entire code (Essential).

SY_ME15: Estimated total number of commented-out lines of code

Enter the estimated total number of commented-out lines of code within the code base (CodeCheck rule CPP_C000 set to 'aggressive').

5 Architecture

5.1 Overview

This tab captures the risk factors associated with discernable, as-built software architectural characteristics of the system. A software architecture is the set of structures needed to reason about the system, including software elements, relations among them, and properties of both [Bass 2013]. Although every software system has an architecture, whether known and documented or not, certain architectural characteristics tend to make a system more understandable, maintainable, testable, and reusable. More specifically, architectural characteristics such as modularity, dependencies, and pattern utilization are often discernable from source code, though a complete software architecture is very difficult to reconstruct only from the code. Hence, the CREW architectural analysis attempts only to discern the degree to which the code base embodies an architectural structure more likely to enhance its sustainability over the system's lifecycle.

Thus, to achieve a good score in the CREW Architecture tab, a codebase should exhibit a high level of code modularity and module cohesiveness. The code should be clearly separated into components that have an unambiguous purpose, and the components should be relatively self-contained. The code should also have a low level of dependencies, especially bi-directional and cyclic dependencies. Cycles are bad in an architecture because they are an unnecessary coupling that makes the code difficult to change. Instead, software components should minimize couplings between each other and avoid complex couplings. A low volume of modules and inter-dependencies should translate to a good score. Lastly, the codebase should have a high level of adherence to a known architectural pattern (layered, client-server, etc.).

SA tools are often used to help analysts ascertain the quality of a system's software architecture. For example, the number and types of interdependencies among modules or classes can be analyzed using Dependency Graphs, a Dependency Structure Matrix (DSM), or other visualizations provided by a SA tool. Visualizations can help analysts determine the number of couplings between components and whether there exist any bidirectional or cyclic dependencies. The Architecture tab is shown in Figure 5.1.1.

Figure 5.1.1 The Architecture Tab

Focus	Risk	W	ID	I	Measurement / Question	A	Finding	D
MODULARITY	1.0	0.333	AR_MD01	0	Atom count (# elements in the system)		13896	0
			AR_MD02	1	Is there a clear separation of functional components within the system?	Yes		0
			AR_MD03	1	Is there a high degree of coupling among components?	No		0
			AR_MD04	1	Within components is there a high degree of coupling among modules?	No		0
			AR_MD05	1	System stability (% of elements (on avg) that would not be affected by a change to an element)		99.9%	0
			AR_MD06	1	Minimum component stability		99.6%	0
			AR_MD07	1	Connectedness (% ordered pairs of systems (A,B) such that A directly or indirectly depends on B)		0.1%	0
			AR_MD08	1	To what degree are global variables utilized within the system?	None		0
DEPENDENCY	2.9	0.333	AR_DP01	1	From the dependency graph, how many dependencies are bidirectional?	Some		0
			AR_DP02	1	How many dependencies are there on COTS or FOSS components?	Many		0
			AR_DP03	1	Total number of internal dependencies (total # of edges between all atoms)		29871	0
			AR_DP04	1	Average dependency (total edges / total # atoms)		2.15	0
			AR_DP05	1	Architectural complexity (# atoms multiplied by # edges / 10.0E+06)		41.51	0
			AR_DP06	1	System cyclicality (% elements in the system that are part of a cycle)		1.0%	0
			AR_DP07	1	Intercomponent cyclicality (% elements in a cycle with systems in other components of the arch)		0.1%	0
PATTERNS	5.0	0.333	AR_PT01	1	Does the system use a common architectural pattern (layering, client-server, MVC, etc.)?	No		0
			AR_PT02	0	Are any COTS or FOSS components cleanly separated from custom application code?	C No		0
			AR_PT03	0	Does the system use the most commonly found architectural pattern: layering?	C No		0
			AR_PT04	0	If layered, is the system strictly layered (calls are only to the adjacent lower layer)?	C Somewhat		0
			AR_PT05	0	Is the architectural pattern used evident in file and subfolder naming and organization?	C Somewhat		0
			AR_PT06	1	Do architectural boundaries exist that hide implementation details within components?	No		0
RISK	2.95							

The following sections describe how to score the CREW Architecture tab, although it should be said that analyzing Architectural Complexity (AC) generally requires an experienced analyst:

5.2 Getting Started

Create an Understand project for the codebase or import the one created by the CREW build master. If it is a subset of a larger codebase, the analyst may wish to create a second project with a name postfixed "_Full" that covers the entire codebase. Unless stated otherwise, Architecture tab questions are to be answered using the _Full project if one exists.

In Understand, create and review a dependency graph (with all dependency types enabled) using Graphs, Dependency, By Folder. Assess and expand the nodes in the graph.

Create a Lattix Architect project for the Understand project (both, if applicable, using the same _Full postfix). Use the "Embedded" option and select the appropriate Understand project folder to create it.

In Lattix, create and review a Dependency Structure Matrix (DSM) and a Conceptual Architecture Diagram (CAD). Apply Component Partitioning at \$root level to the DSM and the CAD from the toolbar to obtain a component partitioned DSM and CAD. Assess and expand the elements within the DSM and CAD. In Lattix, create reports using the main menu selection Reports | Metrics Reports, and then check System Metrics and Architecture Metrics. Access these reports within Lattix.

In Lattix, create a global variable report using main menu selection Reports | Global Variables Report. Access these reports within Lattix and export a copy as a text file.

5.3 Modularity

AR_MD01 Atom count (# elements in the system)

Open the Lattix System Metric Report and enter the value for "Total elements". This is the sum of the header and implementation files, again using the _Full codebase Lattix project if one exists.

Note that this value is unscored within CREW but is used for some auto-calculated metrics which are scored.

AR__MD02 Is there a clear separation of functional components within the system?

EXEMPLAR TOOL(S):

Lattix Architect

PROCESS:

Review top-level Component Partitioned DSM and CAD and capture each as a .png image. Look for dependency counts and number of cyclic dependencies (above the diagonal) between components in the DSM.

EVALUATION:

Review the top-level component partitioned DSM and use the guidance below to answer.

SCORES:

No	Dependency counts are scattered uniformly throughout the DSM and 10 or more cyclic dependency instances are evident.
Somewhat	Dependency counts are spread through the DSM but tend towards lower-triangular and cyclic dependency instances are less than 10.
Yes	Most dependency counts are near the bottom of a lower-triangular DSM with no cyclic dependencies evident.

AR__MD03 Is there a high degree of coupling among components?

EXEMPLAR TOOL(S):

Lattix Architect

PROCESS:

Review top-level component partitioned DSM in Lattix and assess the dependency counts among the components. Also review the Lattix Coupling architecture metric where typical values are in the range [2.0, 6.0] per the Lattix paper on metric values for a sample of well-known systems (Nant, Ant, Apache Server, Spring, Eclipse, .NET Framework). Then answer appropriately.

EVALUATION:

Review the DSM and Coupling metric and use the guide below to answer.

SCORES:

No	Dependency counts are low throughout the DSM and the Coupling metric is less than 1.0.
Few	Dependency counts are mostly low valued within the DSM and the Coupling metric is in [1.0, 4.0].
Some	Dependency counts are mixed low and high valued within the DSM and the Coupling metric is in [4.0, 6.0].

Many	Dependency counts have many high values within the DSM and the Coupling metric is in [6.0, 8.0].
Yes	Dependency counts are high valued within the DSM and the Coupling metric is greater than 8.0.

AR__MD04 Within components is there a high degree of coupling among modules?

EXEMPLAR TOOL(S):
Lattix Architect

PROCESS:

Review top-level DSM in Lattix. Use Component Partitioning, then expand a sampling of at least 25% of the components, especially the largest by SLOC (shown as % of system SLOC on the diagonal). Look for dependency counts and cyclic dependencies within the expanded components which may include sub-components and / or source modules.

EVALUATION:

Review the expanded components within the DSM and use the guide below to answer.

SCORES:

No	Dependency counts are low throughout all expanded components.
Few	Dependency counts are low throughout most expanded components.
Some	Dependency counts are mixed among some expanded components.
Many	Dependency counts are high among most expanded components.
Yes	Dependency counts are high among virtually all expanded components.

AR__MD05 System stability (% of elements (on avg) that would not be affected by a change to an element)

EXEMPLAR TOOL(S):
Lattix Architect

PROCESS:

Review Lattix architecture report and utilize the System Stability metric.

EVALUATION:

Enter the Lattix System Stability architecture metric where typical values are in the range [90.0, 99.0].

AR__MD06 Minimum component stability

EXEMPLAR TOOL(S):

Lattix Architect

PROCESS:

Review Lattix architecture report. Expand the \$root top-level to show component levels within the report.

EVALUATION:

Expand the \$root line for the Lattix architecture metrics and sort on the System Stability column to report the minimum value.

AR__MD07 Connectedness (% ordered pairs of systems (A, B) such that A directly or indirectly depends on B)

EXEMPLAR TOOL(S):

Lattix Architect

PROCESS:

Review Lattix architecture report.

EVALUATION:

Enter the Lattix Connectedness architecture metric where typical values are in the range [2.0, 15.0] per the Lattix metric values for well-known systems.

AR__MD08 To what degree are global variables utilized within the system?

EXEMPLAR TOOL(S):

Lattix Architect

PROCESS:

Review the Lattix global variables report. Export this report and then open in an editor and use a file search on the whole word “Variable” to get the total Count. Note that Lattix defines a global variable as any variable that is a) not of function scope or b) a class variable that is not private. Both situations are deemed undesirable within a code base. Other definitions of a global variable certainly exist. However, we use the more expansive Lattix definition for this question.

EVALUATION:

Obtain a count value from the Lattix Global Variables report.

SCORES:

None	count = 0
Some	count < (PSLOC / 1000)
Many	count >= (PSLOC / 1000)

5.4 Dependency

AR_DP01: From the dependency graph, how many dependencies are bidirectional?

EXEMPLAR TOOL(S):
Understand

PROCESS:

Review top-level dependency graph in Understand and capture as a .png image. Assess the relative number of bidirectional (red) edges versus unidirectional (blue) ones, noting the count values shown for each. Sum the count values for each and obtain a total count by summing the two sums.

EVALUATION:

Review the expanded components within the DSM and use the guide below to answer.

SCORES:

None	No bi-directional dependencies are evident
Some	Less than or equal to 25% of bi-directional dependencies are evident.
Many	More than 25% of the total dependencies are bi-directional

AR_DP02: How many dependencies are there on COTS or FOSS components?

EXEMPLAR TOOL(S):
Lattix, Understand, grep.

PROCESS:

Review top-level component partitioned DSM, CAD, and dependency graph for the codebase to identify commonly known COTS / FOSS components. Apply a grep / search tool (e.g., dnGrep or Sublime Text's find) over the entire codebase to search for terms (using both whole words / non-whole words, and non-case sensitive) identified in the DSM, CAD, and dependency graph, as well as for generic COTS / FOSS terms including: SourceForge, open, license, library, URL, http://, generated, gnu. Review the search matches for frequency and inspect at least 10% of usage of the matches for these terms assessing “are they identifying COTS / FOSS usage?”.

EVALUATION:

After review of all inputs answer as follows:

SCORES:

None	No COTS/FOSS dependencies are evident
Some	COTS/FOSS dependencies are less than or equal to 5% of total dependencies
Many	More than 5% of the total dependencies involve COTS/FOSS

AR__DP03: Total number of internal dependencies (total # of edges between atoms)

Enter the Lattix Internal Dependencies architecture metric where typical values are in the range [1000, 10000].

AR__DP04: Average dependency (total edges / total # atoms)

Value is auto-calculated in CREW. Typical values are in the range [1, 8].

AR__DP05: Architectural complexity (# atoms multiplied by # edges / 10.0E+06)

Value is auto-calculated in CREW. Typical values are in the range [0.1, 1.0].

AR__DP06: System cyclicity (% elements in the system that are part of a cycle)

Enter the Lattix System Cyclicity architecture metric where typical values are in the range [1, 10].

AR__DP07: Intercomponent cyclicity (% elements in a cycle with systems in other components of the arch)

Enter the Lattix Intercomponent Cyclicity architecture metric where typical values are in the range [1, 8].

5.5 Patterns

AR__PT01: Does the system use a common architectural pattern (layering, client-server, MVC, etc.)?

EXEMPLAR TOOL(S):

Lattix, grep

PROCESS:

Most systems use architectural layering. This should be evident in the DSM and CAD views of the system after applying Component Partitioning at \$root level to the codebase. The base layers will be at the DSM's / CAD's bottom, showing that most of the code depends upon layers towards the bottom.

For other architectural patterns, use a grep file search utility to search for certain pattern terms across the codebase. Search for terms including:

bus, message, mqtt, service, client, server, mvc, model, view, controller, event, pub, sub, pub-sub, publish, subscribe, topic, rest, soap, master, slave, database, sql, plug-in, kernel, command, query, proxy, adapter, bridge, state, strategy, observer, api, layer.

Look at search matches for frequency and review the usage of any of these terms in the code: e.g. Are they referring to architectural constructs in the code or merely used sporadically or just in the comments? Note that more than one pattern may exist within a given code base.

EVALUATION:

After review of all inputs answer as follows:

SCORES:

No	No evidence found via Lattix or via search terms
Yes	Evidence found via Lattix or via search terms of one or more patterns

AR__PT02: Are any COTS or FOSS components cleanly separated from custom application code?

EXEMPLAR TOOL(S):

Understand, grep.

PROCESS:

For the components identified in AR__DP02, review the associated files and source code. Look to answer these questions:

- Is custom code isolated from COTS / FOSS code within any one file containing both?
- Are the files for any larger COTS / FOSS components contained within well-named sub-folders?
- Are calls to COTS / FOSS component APIs commented or clearly identified as such?

EVALUATION:

After review of the above answer as follows:

SCORES:

No	No to any of the questions above is prevalent throughout the code
Somewhat	No to any questions above is seen sporadically throughout the code
Yes	Yes to all questions above across the codebase

AR__PT03: Does the system use the most commonly found architectural pattern: layering?

EXEMPLAR TOOL(S):

Understand, Lattix.

PROCESS:

Review the top-level component partitioned DSM and dependency graphs for the codebase and look for components that are depended upon by many application (higher) level components. These will usually be at the bottom of a system or subsystem within the DSM.

EVALUATION:

Answer whether you see a DSM and / or dependency graphs that show the pattern described above.

SCORES:

No	No evidence of layered dependencies is apparent.
Yes	Evidence of layered dependencies is apparent.

AR__PT04: If layered, is the system strictly layered (calls are only to the adjacent lower layer)?

EXEMPLAR TOOL(S):

Lattix.

PROCESS:

Review top-level component partitioned DSM for the codebase and look for sequentially chained dependencies to the lower components (rows).

EVALUATION:

After review of the above answer as follows:

SCORES:

No	Ample evidence exists of calls through multiple layers.
Somewhat	There are some but few instances of calls through multiple layers.
Yes	No evidence of calls through multiple layers.

AR__PT05: Is the architectural pattern used evident in file and subfolder naming and organization?

EXEMPLAR TOOL(S):

Lattix.

PROCESS:

For the patterns identified in AR__PT01, review the component partitioned DSM and the Project Browser and / or CC versus SLOC heatmap in Understand looking for names of folders and files that correlate with the pattern(s) discovered, e.g., are there subfolders with "client" and "server" in their name if the client-server pattern applies?

EVALUATION:

After review as per the process above, answer as follows:

SCORES:

No	No correlation is evident between files/folders and the pattern.
Somewhat	Some correlation is evident between files/folders and the pattern.
Yes	High correlation is evident between files/folders and the pattern.

AR__PT06: Do architectural boundaries exist that hide implementation details within components?

EXEMPLAR TOOL(S):

Lattix.

PROCESS:

For the search results in AR__PT01, review source code in Understand looking for the prevalence of calls made via plug-ins, API's, message handlers, etc.

EVALUATION:

After review as per the process above, answer as follows:

SCORES:

No	No component boundary mechanism was evident.
Somewhat	Some use of component boundary mechanisms was evident.
Yes	Component boundary mechanisms are used throughout.

6 Standards

6.1 Overview

This tab analyzes the degree to which the codebase follows coding standards. Coding standards are collections of coding rules, guidelines, and best practices that help to ensure code quality. Adherence to standards will greatly improve the likelihood that the resulting software embodies the most commonly important quality attributes. For example, adhering to standards can help ensure the system is maintainable, consistent, understandable, safe, secure, testable, and reliable throughout its intended life.

To receive a satisfactory score in this tab, a codebase should exhibit good style, and have a high level of adherence to a chosen internal or published stylistic coding standard. Code that is written clearly and formatted consistently tends to demonstrate good style. The code should also have a high level of adherence to critical-system coding guidelines (MISRA, CERT, etc.). This helps assure the code is predictable and the possibility of crashes is minimized. Lastly, the code should exhibit a high level of adherence to widely accepted language guidelines. If the standards and accepted best practices for the language are followed consistently, this can be indicative of low-risk code.

While SA tools are often useful for identifying some violations, manual inspection of files is often required. Because CREW is designed such that analysts can complete an analysis with just the source code and no supporting documents, the stylistic questions do not prescribe a specific coding style. This is done to gauge whether that any coding style chosen is used consistently. If completing an internal analysis, it may be useful to check for compliance to the project's prescribed coding standard. Note that many standards for safety-critical systems exist, and CREW checks for compliance to a small number of basic criteria common amongst all standards. CREW's approach is modeled after NASA JPL's Power of Ten. The Standards tab is shown in Figure 6.1.1.

Figure 6.1.1 The Standards Tab

Focus	Risk	W	ID	I	Measurement / Question	A	Finding	D	Note
STYLISTIC	1.3	0.100	ST_ST01	1	Is each source file limited to one class or collection of routines with one and only one purpose?		Few	0	
			ST_ST02	1	Do source file headers explain their function?		Some	0	
			ST_ST03	1	Are includes/imports included together?		Yes	0	
			ST_ST04	1	Are methods declared and defined in a logical order?		Yes	0	
			ST_ST05	1	Are C++ overloaded methods grouped together?	C	Yes	0	
			ST_ST06	1	Are only spaces used for whitespace?		Yes	0	
			ST_ST07	1	Are indentations and brace usage consistent?		Yes	0	
			ST_ST08	1	Does there appear to be a column limit in place for source files?		Yes	0	
			ST_ST09	1	Are 'paragraphs' used to separate the code into logically related sections?		Yes	0	
			ST_ST10	1	Are class / module names meaningful?		Yes	0	
			ST_ST11	1	Are class / module names formatted consistently?		Yes	0	
			ST_ST12	1	Are method / function names meaningful?		Yes	0	
			ST_ST13	1	Are method / function names formatted consistently?		Yes	0	
			ST_ST14	1	Are variable names meaningful?		Yes	0	
			ST_ST15	1	Are variable names formatted consistently?		Yes	0	
			ST_ST16	1	Are constant names meaningful?		Yes	0	
			ST_ST17	1	Are constant names formatted consistently?		Yes	0	
			ST_ST18	1	Are test methods / functions consistently marked as such?		Yes	0	
			ST_ST19	1	Do files include a file prologue that explains the purpose of the file?		Yes	0	
			ST_ST20	1	Do methods / functions include a prologue comment block?		Yes	0	
			ST_ST21	1	Are comments formatted consistently?		Yes	0	
			ST_ST22	1	Are units for variables included?		Yes	0	
			ST_ST23	1	Are comments used to mark the ends of code blocks?		Yes	0	
			ST_ST24	1	Are error values formatted consistently?		Yes	0	
CRITICALITY	2.7	0.450	ST_CR01	1	Are goto, continue, or other control flow jumps present within the code?		Few	0	
			ST_CR02	1	Is direct or indirect recursion utilized within the code?		Many	0	
			ST_CR03	1	Do loops always have a fixed upper bound?		Some	0	
			ST_CR04	1	Is dynamic heap memory allocated after initialization?		Many	0	
			ST_CR05	1	Are there functions longer than 60 lines?		No	0	
			ST_CR06	1	Does the assertion density average to minimally two assertions per function?		Yes	0	
			ST_CR07	1	Assertions per KSLOC of source code		0.3333	0	
			ST_CR08	1	Are assertions free of side effects?		Yes	0	
			ST_CR09	1	Do assertions have failure handling?		Yes	0	
			ST_CR10	1	Are data objects declared near their first use?		Yes	0	
			ST_CR11	1	Is the preprocessor used beyond includes, guards, and simple macros?		Yes	0	
			ST_CR12	1	Is there evidence that the code compiles with zero (0) warnings?		Yes	0	
			ST_CR13	1	Are function return values checked consistently?		Yes	0	
			ST_CR14	1	Does the code utilize multi-level pointer dereferences?		Yes	0	
			ST_CR15	1	Does the code utilize function pointers?		Yes	0	
			ST_CR16	1	Are data objects declared at the smallest possible level of scope?		Yes	0	
LANGUAGE C/C++	1.5	0.450	ST_LC01	1	Are goto, continue, or other control flow jumps present within the code?		Yes	0	
			ST_LC02	1	Do switch statements have a default case?		Yes	0	
			ST_LC03	1	Are case fall-throughs marked with a comment?		Yes	0	

The following sections describe how to score the Standards tab of CREW:

6.2 Getting Started

Sampling

When sampling files it is imperative to examine as wide a berth as possible. At least 10% of all project files should be examined. In addition, care should be taking to sample files from every module and submodule in the codebase to insure consistency throughout.

Scoring

Many questions in the Standards section are answered with a drop-down selection. As a general guideline, here are the standard thresholds for which selection to choose for most questions in the Standards tab. Questions that deviate from these thresholds will list their own threshold table.

Selection	Sampled Elements with Violation
All/Yes	None or a small handful
Many/Most	< 25%
Somewhat	< 50%

Few/Seldom	< 80%
No	80% or higher

Note that these are guidelines and scores, particularly near boundary points, must be judged by the analyst.

6.3 Stylistic

ST__ST01: Is each source file limited to one class or collection of routines with one and only one purpose?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Ensure each file contains only one class. If the file contains only routines, then ensure all routines have a common link between them, i.e., all functions involve encryption or all functions pertain to the manipulation of a data object.

SCORING: Use standard thresholds.

ST__ST02: Do source file headers explain their function?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Ensure the headers of files include a description of what the class/routines within are meant to do.

SCORING: Use standard thresholds.

ST__ST03: Are includes/imports included together?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Ensure all includes/imports are listed together.

SCORING: Use standard thresholds.

ST__ST04: Are methods declared and defined in a logical order?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: See if functions are ordered alphabetically or if related functions are grouped together.

SCORING: Use standard thresholds.

ST_ST05: Are C++ overloaded methods grouped together?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a tool that allows the user to view a list of all functions.

EVALUATION: Find any overloaded functions and then assure they're grouped together.

SCORING: Use standard thresholds.

ST_ST06: Are only spaces used for whitespace?

EXEMPLAR TOOL(S): grep or any tool with search functionality

PROCESS: Search the project for any tabs.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Tabs Found
All/Yes	None or a small handful
Many/Most	< 0.5 per ksloc
Somewhat	< 5 per ksloc
Few/Seldom	< 25 per ksloc
No	25+ per ksloc

ST_ST07: Are indentations and brace usage consistent?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Ensure indentations are consistent. Are all braces on the next line or at the end of the line?

SCORING: Use standard thresholds.

ST__ST08: Does there appear to be a column limit in place for source files?

EXEMPLAR TOOL(S): Manual inspection / editor

PROCESS: Sample files.

EVALUATION: Many editors will allow the user to see a “map” of the current files. This can be useful to see if there are any standout long lines.

SCORING: Use standard thresholds.

ST__ST09: Are ‘paragraphs’ used to separate the code into logically related sections?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Ensure that blank lines are used to separate the code into related chunks. For instance, if three related variables are being assigned together, they should be separated by the surrounding code.

SCORING: Use standard thresholds.

ST__ST10: Are class / module names meaningful?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a tool that allows the user to see a list of all classes or modules. The analyst can then scroll down the list to check for meaningfulness.

EVALUATION: Names should be unambiguous and give the reader insight into the classes purpose.

SCORING: Use standard thresholds.

ST__ST11: Are class / module names formatted consistently?

EXEMPLAR TOOL(S): Manual inspection, linter

PROCESS: Sample files.

EVALUATION: See what the most common style for classes or modules is. Then, use a linter and check for compliance the commonly used style.

SCORING: Use standard thresholds.

ST__ST12: Are method / function names meaningful?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a tool that allows the user to see a list of all methods or functions. The analyst can then scroll down the list to check for meaningfulness.

EVALUATION: Names should be unambiguous and give the reader insight into the functions purpose.

SCORING: Use standard thresholds.

ST__ST13: Are method / function names formatted consistently?

EXEMPLAR TOOL(S): Manual inspection, linter

PROCESS: Sample files.

EVALUATION: See what the most common style for functions or methods is. Then, use a linter and check for compliance the commonly used style.

SCORING: Use standard thresholds.

ST__ST14: Are variable names meaningful?

EXEMPLAR TOOL(S): Manual inspection

PROCESS: Sample files.

EVALUATION: Names should be unambiguous and give the reader into the purpose of the variable. Single letter names should be avoided outside of loop counters.

SCORING: Use standard thresholds.

ST__ST15: Are variable names formatted consistently?

EXEMPLAR TOOL(S): Manual inspection, linter

PROCESS: Sample files.

EVALUATION: See what the most common style for variables. Then, use a linter and check for compliance the commonly used style.

SCORING: Use standard thresholds.

ST__ST16: Are constant names meaningful?

EXEMPLAR TOOL(S): Manual inspection

PROCESS: Sample files.

EVALUATION: Names should be unambiguous and give the reader into the purpose of the constant.

SCORING: Use standard thresholds.

ST__ST17: Are constant names formatted consistently?

EXEMPLAR TOOL(S): Manual inspection, linter

PROCESS: Sample files.

EVALUATION: See what the most common style for variables and constants is. Then, use a linter and check for compliance the commonly used style.

SCORING: Use standard thresholds.

ST__ST18: Are test methods / functions consistently marked as such?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a tool that allows the user to see a list of all methods or functions.

EVALUATION: Scroll down and check for any functions marked as tests.

EVALUATION: Use standard thresholds.

ST__ST19: Do files include a file prologue that explains the purpose of the file?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Check for prologues. Such prologues should include the class name or the common sphere of the included routines as well as a list of all functions/methods included in the file.

SCORING: Use standard thresholds.

ST__ST20: Do methods / functions include a prologue comment block?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample functions.

EVALUATION: Check for prologues. Such prologues should include a description of the method/function's purpose, its expected input, its expected output, and any potential errors it may return.

SCORING: Use standard thresholds.

ST__ST21: Are comments formatted consistently?

EXEMPLAR TOOL(S): grep

PROCESS: Using a tool or a script, search project files for “//” and “/*” comments.

EVALUATION: Ensure only one is used within each file

SCORING: Use standard thresholds.

ST_ST22: Are units for variables included?

EXEMPLAR TOOL(S): Manual inspection, grep.

PROCESS: A good starting point is to search the project for common units (feet, miles, seconds, meters, etc.), or common measurements (speed, distance, degrees, time, etc.). Additionally, sample files and check for variables referring to measurements.

EVALUATIONS: Variables that represent measurements should include units either within the variable name or as a comment.

SCORING: Use standard thresholds.

ST_ST23: Are comments used to mark the ends of code blocks?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Check code blocks for ending comments. These may take the form of comments such as “//f” to end if statements or a comment containing the full condition of the code block such as “// if (condition == true)”.

SCORING: Use standard thresholds.

ST_ST24: Are error values formatted consistently?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample functions.

EVALUATION: Ensure all error values follow the same format.

SCORING: Use standard thresholds.

6.4 Criticality

Criticality concerns adherence to standards for code that is deemed safety-critical or mission-critical. The main goal of these standards is to make code as predictable, simple, and verifiable as possible. Many of these rules can be checked for by a static analyzer. Most analyzers are capable of this.

For example, the following is an example configuration of rules in SciTool’s Understand CodeCheck feature that will be useful in answering these questions.

```

{
  "automatic": false,
  "checks": [
    "AC_01",
    "EFFECTIVECPP_26",
    "MISRA08_0-1-7",
    "MISRA08_18-4-1",
    "MISRA08_5-0-19",
    "MISRA12_8.9",
    {
      "id": "RECOMMENDED_04",
      "options": {
        "CodeOnly": true,
        "LineLimit": 60
      }
    },
    "RECOMMENDED_06",
    "RECOMMENDED_17"
  ],
  "excludes": [
  ],
  "name": "StandardsCriticality",
  "parents": [
  ]
}

```

ST_CR01: Are goto, continue, or other control flow jumps present within the code?

EXEMPLAR TOOL(S): Understand, grep

PROCESS Many static analyzers can check for these or searching project files for “goto” will also reveal any violations.

EVALUATION: Ensure goto statements and other control flow jumps are not present within the code. Exceptions are often made in certain instances, such as goto statements found in board support packages or other vendor supplied files.

SCORING:

Selection	Number of Violations Found
All/Yes	10+
Many/Most	< 10
Somewhat	< 5

Few/Seldom	< 2
No	None

ST__CR02: Is direct or indirect recursion utilized within the code?

EXEMPLAR TOOL(s): Understand

PROCESS: Use a static analyzer to check for cases of recursion.

EVALUATION: Manually confirm functions are recursive.

SCORING:

Selection	Number of Violations Found
All/Yes	10+
Many/Most	< 10
Somewhat	< 5
Few/Seldom	1
No	None

ST__CR03: Do loops always have a fixed upper bound?

EXEMPLAR TOOL(S): Understand, grep

PROCESS: Use a static analyzer to check for modification of loops or lack of fixed upper bounds as well as search the project for instances of looping to sample.

EVALUATION: All loops should have a fixed upper bound that can never be exceeded.

SCORING: Use standard threshold.

ST__CR04: Is dynamic heap memory allocated after initialization?

EXEMPLAR TOOL(S): Understand

PROCESS: Use static analyzer to check for dynamic memory usage.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	5+ per ksloc
Many/Most	< 5 per ksloc
Somewhat	< 2.5 per ksloc
Few/Seldom	< 0.5 per ksloc
No	None or a small handful

ST__CR05: Are there functions longer than 60 lines?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to ensure all function are less than 60 lines of code.

EVALUATION: Verify instances are true positives.

SCORING: Use standard thresholds.

ST__CR06: Does the assertion density average to at minimum two assertions per function?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for asserts. Most projects will have their own wrappers for asserts. Compile a list of all of them and then search the project to find the total number of asserts used and then divide by the number of functions.

EVALUATION: The resulting density should be at least 2.

SCORING: Yes or No.

ST__CR07: Assertions per KSLOC of source code

EXEMPLAR TOOL(S): grep

PROCESS: Search project for asserts. Most projects will have their own wrappers for asserts. Compile a list of all of them and then search the project to find the total number of asserts used. then divide by the KSLOC to calculate the assertion density.

EVALUATION: Verify instances are true positives.

SCORING: Input the calculated assertion density.

ST_CR08: Are assertions free of side effects?

EXEMPLAR TOOL(S): grep

PROCESS: Search for asserts and check for any possible side effects.

EVALUATION: Assertions must be free of side effects, i.e., the code should be able to run with the assert statement commented out. Below is an example of a compliant and non-complaint assert:

```
int m_variable = 0;
int m_limit = 10;

bool foo(int value) {
    bool result = false;
    if(value < 100){
        m_variable = value;
        result = true;
    }
    return result;
}

bool bar (int value){
    bool result = false;
    if(value < m_limit){
        result = true;
    }
    return result;
}

// non compliant
assert(foo(x));

//compliant
bool returnValue;
returnValue = foo(x);
assert(returnValue);

//compliant
assert(bar(x));
```

The first assert statement is not compliant as foo(), in addition to returning a bool, assigns a value to a member variable. The following assert shows how to make the statement compliant. By pulling the call to foo() out of the assert statement, the assert statement itself can still be removed without affecting the behavior of the code. The third statement is also compliant as bar() only returns a value and does not have any other side effects.

SCORING: Use standard thresholds.

ST__CR09: Do assertions have failure handling?

EXEMPLAR TOOL(S): grep

PROCESS: Search for asserts and check for error handling.

EVALUTION: Assertions should be able to be recovered from.

SCORING: Use standard thresholds.

ST__CR10: Are data objects declared near their first use?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for postponable declarations.

EVALUATION: Verify instances are true positives.

SCORING: Use standard thresholds.

ST__CR11: Is the preprocessor used beyond includes, guards, and simple macros?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for function-like macros and other more complicated constructs.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	100+
Many/Most	< 100
Somewhat	< 50

Few/Seldom	< 10
No	None or a small handful

ST__CR12: Is there evidence that the code compiles with zero (0) warnings?

EXEMPLAR TOOL(S): None:

PROCESS: Check build output for warnings.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No.

ST__CR13: Are function return values checked consistently?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to check for unused returns.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None or a small handful
Many/Most	< 0.5 per ksloc
Somewhat	< 2.5 per ksloc
Few/Seldom	< 5 per ksloc
No	5+ per ksloc

ST__CR14: Does the code utilize multi-level pointer dereferences?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to check for high levels of indirection.

EVALUATION: Confirm results as true positive. Exceptions should be made for things such as 3d arrays which many analyzers will flag.

SCORING:

Selection	Number of Violations Found
All/Yes	50+
Many/Most	< 50
Somewhat	< 10
Few/Seldom	< 5
No	None

ST_CR15: Does the code utilize function pointers?

EXEMPLAR TOOL(S): Regex search.

PROCESS: Use regex to search for any function pointers.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

ST_CR16: Are data objects declared at the smallest possible level of scope?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for objects with too wide a scope.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

6.5 Language C/C++

The Language focus concerns proper use of the used programming language. Every programming has dos, don'ts, and best practices. Failure to follow these standards can result in code that is prone to errors, confusion, and unpredictable behavior. Many of the questions in this section can be checked with a static analyzer. Below is an example of a configuration for SciTool's Understand CodeCheck feature to check for proper language utilization.

```

{
  "automatic": false,
  "checks": [
    "A7-1-7",
    {
      "id": "CPP_C036",
      "options": {
        "allowInNested": false
      }
    },
    "CPP_V001",
    "EFFECTIVECPP_26",
    "M6-2-1",
    "MISRA12_16.4",
    "RECOMMENDED_06"
  ],
  "excludes": [
  ],
  "name": "Language C",
  "parents": [
  ]
}

```

ST_LC01: Are goto, continue, or other control flow jumps present within the code?

EXEMPLAR TOOL(S): Understand

PROCESS: Many static analyzers can check for these or searching project files for “goto” will also reveal any violations.

EVALUATION: Exceptions are often made in certain instances, such as goto statements found in board support packages.

SCORING:

Selection	Number of Violations Found
All/Yes	10+
Many/Most	< 10
Somewhat	< 5
Few/Seldom	< 2
No	None

ST__LC02: Do switch statements have a default case?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to check for switch statements without default cases.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None
Many/Most	< 2/ksloc
Somewhat	< 5/ksloc
Few/Seldom	< 10/ksloc
No	10+ per ksloc

ST__LC03: Are case fall-throughs marked with a comment?

EXEMPLAR TOOL(S): Regex search

PROCESS: The codebase can be searched for all switch statements, or a regex search can be used to find consecutive case statements. Any fall-throughs should include a comment to verify that it was an intentional design.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	All fall throughs marked or none found
Many/Most	Most fall throughs commented with 1 or 2 outliers
Somewhat	Several uncommented fall throughs
Few/Seldom	Most fall throughs uncommented with 1 or 2 outliers
No	No fall throughs commented

ST__LC04: Is each statement on its own line?

EXEMPLAR TOOL(S): Regex search

PROCESS: Use a regular expression to search for lines with multiple statements.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None
Many/Most	< 2/ksloc
Somewhat	< 5/ksloc
Few/Seldom	< 10/ksloc
No	10+ per ksloc

ST__LC05: Are variables declared on their own line?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to check for multiple variables being declared on the same line.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None
Many/Most	< 10
Somewhat	< 50
Few/Seldom	< 100
No	100+

ST__LC06: Are variables declared close to their first use?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for postponable declarations.

EVALUATION: Verify instances are true positives.

SCORING: Use standard thresholds.

ST__LC07: Are logic and assignments mixed within the same statement?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for assignments within logic statements or other sub-expressions.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	100+
Many/Most	< 100
Somewhat	< 50
Few/Seldom	< 10
No	None

ST__LC08: Does the code utilize the volatile keyword without a comment documenting why?

EXEMPLAR TOOL(S): grep

PROCESS: Search the project for the volatile key word.

EVALUATION: Ensure each statement is documented with a comment.

SCORING:

Selection	Number of Violations Found
All/Yes	10+
Many/Most	< 10
Somewhat	< 5
Few/Seldom	< 2
No	None

ST_LC09: Are C and C++ constructs mixed unnecessarily?

EXEMPLAR TOOL(S): Understand, cppcheck

PROCESS: Use a static analysis tool to search for cases of mixing of C and C++ constructs such as mixing allocations (using the delete keyword on alloc()'d memory) or mixing C and C++ style casts.

EVALUATION: Verify instances as true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None
Many/Most	< 10
Somewhat	< 50
Few/Seldom	< 100
No	100+

ST_LC10: Are floating point values compared to integers?

EXEMPLAR TOOL(S): grep

PROCESS: Search project files for logic statements.

EVALUATION: Ensure that none contain a floating point being compared to an integer.

SCORING:

Selection	Number of Violations Found
All/Yes	10+
Many/Most	< 10
Somewhat	< 5
Few/Seldom	< 2
No	None

6.6 Language C++

All questions in the Language C++ focus can be answered manually using eyes-on-code, however they are more effectively answered using static analysis tools. Ideally, several static analyzers would be used and have their outputs aggregated for the most complete picture of potential risks in the code base. Additionally, a single question may have several static analyzer checkers that correspond to it within the same tool.

In addressing the following line items, our tool of choice is Parasoft, but any reputable SA tool can be used based on your organizational need. Within each line item below, there is a list of Parasoft checkers that were identified to have analyzed the code for risks and bugs associated with that line item. A test configuration containing those checkers was created and run across the code base, and a post-processing script was used to aggregate each checker's findings and map to a CREW line item. An analyst then selects ~10% of the findings and adjudicates their validity (as SA tools are known to generate false positives). From there, based on the number of violations, the line items are filled in using the table below:

Selection	Number of Violations Found
All/Yes	None or a small handful
Many/Most	< 100
Somewhat	< 500
Few/Seldom	< 1000
No	1000+

A sample of the script output for the mapping of checkers to CREW line items can be seen below .

ST__LP01: Does the code follow the "rule of 3/5" while maintaining uniform scoping?

- AUTOSAR-A12_0_1-a
- AUTOSAR-A1_1_1-b
- CERT_CPP-MEM51-c
- HICPP-12_5_1-a
- JSF-068
- MRM-04
- MRM-05
- MRM-40_b
- MRM-40_c

ST__LP02: Is the implementation of assignment/move operators/constructors safe and within accepted C++ guidelines?

- AUTOSAR-A10_3_5-a
- AUTOSAR-A13_2_1-a
- AUTOSAR-A6_2_1-a
- AUTOSAR-A6_2_1-b

- CODSTA-CPP-21
- CODSTA-CPP-24
- HICPP-8_2_3-a
- HICPP-12_5_3-a
- HICPP-12_5_4-a
- HICPP-12_5_6-a
- HICPP-12_5_7-a
- JSF-077
- JSF-083_a
- JSF-083_b
- JSF-125_b
- OOP-02
- OOP-27

ST_LP03: Are special member functions explicitly declared at the proper scope when required?

- AUTOSAR-A12_8_6-a
- AUTOSAR-A14_5_1-a
- AUTOSAR-M14_5_3-a
- CODSTA-MCPP-06_a
- CODSTA-MCPP-06_b
- HICPP-12_5_2-a
- HICPP-12_5_8-a
- JSF-076
- MRM-37
- MRM-38
- MRM-47
- MRM-48
- OOP-23

- OOP-30

- OOP-38

ST__LP04: Are structs used only as data containers and not as substitutes for classes?

- AUTOSAR-A11_0_1-a

- AUTOSAR-A11_0_2-a

- CODSTA-CPP-10

- JSF-156

ST__LP05: Are identifiers declared such that none are inadvertently hidden?

- AUTOSAR-A10_2_1-a

- AUTOSAR-A10_2_1-b

- HICPP-3_1_1-a

- HICPP-3_1_1-b

- HICPP-3_1_1-c

- HICPP-3_1_1-d

- HICPP-3_1_1-e

- HICPP-9_1_2-a

- MISRA2008-10_3_1

- OOP-04

- OOP-33

- OOP-54

ST__LP06: Are identifiers overloaded with the proper keywords as required?

- HICPP-13_1_1-a

- HICPP-13_1_2-a

- PB-18

- PB-52

- PB-53

ST__LP07: Are virtual functions typed and declared only where circumstances allow?

- AUTOSAR-M10_3_3-a
- HICPP-9_1_5-a
- MISRA2008-10_3_2
- OOP-25

ST__LP08: Are C++ keywords and symbols used properly?

- AUTOSAR-A7_1_5-a
- AUTOSAR-A8_5_3-a
- CODSTA-MCPP-08_a
- CODSTA-MCPP-08_b
- HICPP-7_1_8-a
- AUTOSAR-A12_1_4-a
- AUTOSAR-A10_3_1-a
- HICPP-10_2_1-a
- AUTOSAR-A15_4_4-a
- AUTOSAR-A7_1_2-a
- AUTOSAR-A7_1_2-b
- HICPP-2_5_3-a
- CODSTA-CPP-42
- HICPP-7_3_1-a
- HICPP-7_4_1-a

ST__LP09: Do constructors use initializing methods for their class fields using best practices?

- AUTOSAR-A12_1_2-a
- AUTOSAR-A12_1_3-a
- AUTOSAR-A12_6_1-a
- AUTOSAR-A3_3_2-a

- HICPP-12_4_2-a
- HICPP-12_4_4-a
- INIT-07
- INIT-08
- INIT-09
- INIT-13
- INIT-15

ST__LP10: Do constructors use keywords, overload appropriately, and call other constructors in the inheritance chain?

- AUTOSAR-A12_1_1-a
- AUTOSAR-A8_5_4-a
- CODSTA-CPP-19
- HICPP-12_1_1-a
- HICPP-12_4_5-a
- JSF-077.1
- OOP-14

ST__LP11: Do constructor bodies use allowed functions/variables and initialize the necessary member attributes?

- HICPP-12_4_1-a
- HICPP-12_4_1-b
- JSF-071
- JSF-071_d
- JSF-071_e
- JSF-071_f
- OOP-08
- CODSTA-CPP-40

ST_LP12: Are destructors not called directly and only use identifiers which are safe?

- HICPP-12_4_1-c
- JSF-070.1_b
- JSF-079
- JSF-079_b
- OOP-09

ST_LP13: Is multiple inheritance used minimally and cautiously?

- AUTOSAR-M10_2_1-a
- HICPP-10_1_1-a
- HICPP-10_3_1-a
- JSF-088
- JSF-088_b
- OOP-03
- OOP-05

ST_LP14: Are classes within inheritance chains declared virtual as necessary?

- AUTOSAR-A12_4_1-a
- AUTOSAR-A12_4_2-a
- AUTOSAR-M10_1_2-a
- AUTOSAR-M10_1_3-a
- JSF-088.1
- OOP-39

ST_LP15: Are inheritance chains built properly?

- METRICS-05
- AUTOSAR-A10_4_1-a
- AUTOSAR-M10_1_1-a
- OOP-37

ST_LP16: Are the subscript, unary, and comma operators overloaded minimally and declared appropriately?

- AUTOSAR-A13_5_1-a
- CODSTA-CPP-38
- AUTOSAR-M5_3_3-a
- HICPP-13_2_1-b

ST_LP17: Are the comparison operators overloaded minimally, declared appropriately, and in terms of each other?

- AUTOSAR-A13_5_4-b
- AUTOSAR-A13_5_5-a
- AUTOSAR-A13_5_5-b
- HICPP-13_2_1-a
- HICPP-13_2_2-a

ST_LP18: Are all required variants defined when the new and delete operators are overloaded?

- AUTOSAR-A18_5_4-a
- MRM-14
- MRM-15
- MRM-16
- MRM-17

ST_LP19: Are the arithmetic operators overloaded minimally, declared appropriately, and in terms of each other?

- CODSTA-CPP-07
- CODSTA-CPP-28
- CODSTA-CPP-29
- HICPP-13_2_2-b
- HICPP-13_2_3-a
- HICPP-13_2_5-a
- CODSTA-CPP-30

ST_LP20: Are operators overloaded minimally, returning data as expected, and in terms of other operators?

- CODSTA-CPP-03
- CODSTA-CPP-44
- CODSTA-CPP-20
- CODSTA-CPP-52
- JSF-085
- PB-09

ST_LP21: Is the best available smart pointer and instantiation method used and preferred over raw pointers?

- AUTOSAR-A18_1_3-a
- AUTOSAR-A20_8_4-a
- AUTOSAR-A20_8_5-a
- CODSTA-MCPP-16_a
- CODSTA-MCPP-16_b
- HICPP-17_4_2-a
- HICPP-3_4_3-a

ST_LP22: Are smart pointers initialized only with allowed objects or types?

- AUTOSAR-A18_1_4-a
- AUTOSAR-A20_8_3-a
- AUTOSAR-A20_8_7-a
- CERT_CPP-MEM56-a
- HICPP-17_3_4-a

ST_LP23: Are smart pointers used as function parameters minimally, following best practices?

- AUTOSAR-A8_4_12-a
- AUTOSAR-A8_4_13-b
- AUTOSAR-A8_4_13-c

- AUTOSAR-A8_4_13-a

ST__LC24: Does the code utilize void* without a comment documenting why?

EXEMPLAR TOOL(S): grep

PROCESS: Search the project for the void*.

EVALUATION: Ensure each statement is documented with a comment.

SCORING:

Selection	Number of Violations Found
All/Yes	10+
Many/Most	< 10
Somewhat	< 5
Few/Seldom	< 2
No	None

ST__LP25: Are identifiers not reused within the same namespace?

- AUTOSAR-A2_10_4-a

- AUTOSAR-A2_10_4-b

ST__LP26: Is the global namespace referenced correctly and does it contain only the minimum needed declarations?

- AUTOSAR-M7_3_1-a

- CODSTA-CPP-23

ST__LP27: Are unnamed namespaces and use of the 'using' keyword not utilized in header files?

- CERT_CPP-DCL59-a

- JSF-100

ST__LP28: Are identifiers' definitions/types/declarations separated by namespaces without overlap?

- CODSTA-CPP-26
- CODSTA-CPP-27
- MISRA2008-14_5_1
- MISRA2008-7_3_5

ST__LP29: Are namespaces used and nested minimally, with the standard namespaces remaining unmodified?

- CERT_CPP-DCL58-a
- JSF-099
- OOP-51

ST__LP30: Are streams utilized in a safe manner?

- BD-PB-STREAMFLAGS
- BD-PB-STREAMFMT
- CERT_CPP-FIO50-a

ST__LP31: Does the code follow best practice guidance when using the STL?

- CERT_CPP-CTR57-a
- CERT_CPP-STR53-a
- HICPP-17_1_1-a
- HICPP-17_5_1-a
- OWASP2019-API3-a
- STL-01
- STL-02
- STL-03
- STL-04
- STL-05
- STL-06
- STL-07

- STL-08
- STL-09
- STL-10
- STL-11
- STL-12
- STL-13
- STL-15
- STL-17
- STL-18
- STL-19
- STL-20
- STL-21
- STL-22
- STL-24
- STL-25
- STL-26
- STL-27
- STL-28
- STL-30
- STL-31
- STL-32
- STL-33
- STL-34
- STL-35
- STL-38
- STL-39
- STL-40

- STL-41

- STL-42

ST__LP32: Are the member states of classes and functions immutable by outside entities?

- AUTOSAR-A9_3_1-b

- CODSTA-CPP-06

- CODSTA-CPP-22

- CODSTA-CPP-25

ST__LP33: Are global identifiers encapsulated?

- CODSTA-CPP-18

- JSF-207

- OOP-10

ST__LP34: Are templates used minimally and in a safe manner following best practices?

- AUTOSAR-A14_5_3-a

- AUTOSAR-A14_7_2-a

- AUTOSAR-M14_6_1-a

- CODSTA-CPP-33

- HICPP-14_2_1-a

- HICPP-14_2_2-a

- HICPP-14_2_3-a

- TEMPL-01

- TEMPL-02

- TEMPL-03

- TEMPL-04

- TEMPL-05

- TEMPL-06

ST__LP35: Are iterators uses correctly?

ST__LP36: Are exceptions and their definitions/types used in accordance with their specification?

- AUTOSAR-A15_1_1-a
- AUTOSAR-A15_1_2-a
- AUTOSAR-A15_4_3-a
- CERT_CPP-ERR55-a
- JSF-208

ST__LP37: Are handlers provided for possible exceptions thrown, in order, and called only during normal program flow?

- AUTOSAR-A15_3_4-a
- AUTOSAR-M15_0_3-a
- AUTOSAR-M15_3_7-a
- CERT_CPP-ERR50-k
- CERT_CPP-ERR51-b
- CERT_CPP-ERR58-a
- HICPP-15_3_1-a
- HICPP-15_3_2-b
- OWASP2021-A5-a

ST__LP38: Does the code only throw minimally, where allowed?

- AUTOSAR-M15_1_1-a
- AUTOSAR-M15_1_2-a
- CERT_CPP-ERR50-b
- CERT_CPP-ERR50-e
- CERT_CPP-ERR50-i
- HICPP-15_1_1-b
- HICPP-15_2_1-a
- HICPP-15_3_2-a

- HICPP-15_3_2-c
- HICPP-1_3_5-a
- OWASP2021-A5-b
- OWASP2021-A9-a

ST__LP39: Does the code follow a consistent throw/catch paradigm?

- CERT_CPP-ERR61-a
- CERT_CPP-ERR61-b

7 Maintainability Comments

7.1 Overview

Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [IEEE 1990]. Maintainable code allows others to easily understand, support, modify, and reuse it over its full intended lifetime. “[S]oftware developers now spend about 70% of their coding-related time in understanding the code, while the writing of code only accounts for about 5%” [CISQ 2022]. This tab identifies whether the use of comments within the codebase is consistent, and effectively promotes understanding the underlying code.

To receive a satisfactory score in this tab, there should be a high number of good quality comments within the codebase. When determining the quantity of comments present, boilerplate comments are typically excluded from the analysis. A high comment density and detailed file/module, class, and function headers are indicative of good comment usage. The consistent presence of high-quality commenting that accurately explains the code's purpose and intent is required for a good score.

Many scripts are provided to automate and simplify parts of this analysis. SA tools also play a role in scoring this tab, but the direct visual inspection of source files is often needed to answer many of these questions. In some situations, when there is a wealth of data to analyze (i.e., script output results, all the source files, etc.), sampling is often employed to help narrow and generalize information for input into CREW. In general, there are 2 types of sampling used in this section. The first is a sampling of source files, in those instances a script is used to list based on a set distribution size N (default is 20), the N files with the greatest number of comments, N files in the 50th percentile in terms of comments, and N files with the least number of comments. The analyst will look at these $3*N$ files in total in determining an answer in CREW. The second type of sampling is sampling script output, where the analyst will spot-check ~10% of the script output to generate the answer for CREW. The Maintainability Comments tab is shown in Figure 7.1.1.

Figure 7.1.1 The Maintainability Comments Tab:

Focus	Risk	W	ID	I	Measurement / Question	A	Finding	D	Note
QUANTITY	2.7	0.500	MACOQN01	0	Total number of source files		• 1882	0	
			MACOQN02	0	Total number of (all) lines in the code base		• 631056	0	
			MACOQN03	0	Source lines of code ignoring blank and comment lines (PSLOC)		• 382397	0	
			MACOQN04	0	Number of comment lines in the code base		• 147616	0	
			MACOQN05	1	Uncorrected comment to code ratio		• 39%	0	
			MACOQN06	1	Estimated total number of commented-out lines of code		• 1035	0	
			MACOQN07	0	Approximate total comment lines in existing module header blocks		50	0	
			MACOQN08	1	Corrected comment to code ratio		• 38%	0	
			MACOQN09	1	Estimated percent of source files containing a module header block		90%	0	
			MACOQN10	1	Are existing module header comment blocks merely copyright info?		Yes	0	
			MACOQN11	1	Estimated percent of data declarations that are commented		50%	0	
			MACOQN12	1	Estimated percent of class, function, and method declarations that are commented		50%	0	
			MACOQN13	1	Prevalence of commented-out code (which count as comment lines)		Some	0	
			MACOQN14	1	Overall the quantity of code comments is		Inconsistent	0	
QUALITY	2.8	0.500	MACOQL01	1	Estimated percent of source files with a meaningful preamble comment block re: purpose		75%	0	
			MACOQL02	1	Are code modules separated into standardized sections via comments?		Few	0	
			MACOQL03	1	Are C-style /* ... */ comments used?		Many	0	
			MACOQL04	1	Are // comments used?		Some	0	
			MACOQL05	1	Are end-of-line comments used?		Some	0	
			MACOQL06	1	Are block comments spanning multiple lines used?		Some	0	
			MACOQL07	1	Are comments written at the level of the code's intent, i.e. what and why, not how?		Some	0	
			MACOQL08	1	Are comments used to mark requirements being satisfied in code?		Some	0	
			MACOQL09	1	Are comments used to mark developer changes or modification dates?		Some	0	
			MACOQL10	1	Do comments exist that relate to constructs from an existing SAS or SDS?		Some	0	
			MACOQL11	1	Number of lines containing suspicious comments (todo, hack, fixme, kludge, later, bug, etc.)		100	0	
			MACOQL12	1	Estimated percent of comments that are extraction formatted (Doxxygen, JavaDoc, etc.)		0%	0	
			MACOQL13	0	Are extraction formatted (Doxxygen, etc.) comments used fully and consistently?	C	Few	0	
			MACOQL14	1	Are comments accurate to the code?		Some	0	
			MACOQL15	1	Are comments written at a level to be useful for someone unfamiliar with the code?		Some	0	
			MACOQL16	1	Is the commenting style, level of detail, and formatting consistent throughout?		Somewhat	0	
			MACOQL17	1	Are comments indented the same as the corresponding code?		Most	0	
			MACOQL18	1	Are comments describing subsequent code lines preceded with (at least) one blank line?		No	0	
			MACOQL19	1	Do comments add to or detract from the code's overall understandability?		Add	0	
			MACOQL20	1	Does the commenting style appear easy to maintain?		Somewhat	0	
			MACOQL21	0	Is any portion of the code auto-generated via a code generation tool during the build process?		Yes	0	
			MACOQL22	1	Is auto-generated code during the build process consistently identifiable via a comment signature?	C	No	0	
			MACOQL23	0	Is any portion of the code auto-generated via a code generation tool prior to building?		Yes	0	
			MACOQL24	1	Is any non-build auto-generated code consistently identifiable via a comment signature?	C	Yes	0	
RISK	2.75								

The following sections describe how to score the Maintainability Comments tab of CREW:

7.2 Quantity

MACOQN01: Total number of source files

From System tab, Metrics Focus, Item #8.

MACOQN02: Total number of (all) lines in the code base

From System tab, Metrics Focus, Item #11.

MACOQN03: Source lines of code ignoring blank and comment lines (PSLOC)

From System tab, Metrics Focus, Item #12.

MACOQN04: Number of comment lines in the code base

From System tab, Metrics Focus, Item #14.

MACOQN05: Uncorrected comment to code ratio

From System tab, Metrics Focus, Item #15.

MACOQN06: Estimated total number of commented-out lines of code.

From System tab, Metrics Focus, Item #22.

MACOQN07: Approximate total comment lines in existing module header blocks.

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py using mode one (1) and utilize the output to answer the question.

EVALUATION:

See Figure 7.4.2 for utilization of script output.

SCORES:

Take the direct output as the score.MACOQN08: Corrected comment to code ratio.

Calculated by using the following equation (done automatically in spreadsheet):

$$\frac{TOTAL_COMMENT_LINES - COMMENTED_OUT_LINES_OF_CODE - TOTAL_MODULE_BLOCK_LINES}{PSLOC}$$

MACOQN09: Estimated percent of source files containing a module header block.

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py using mode one (1) and utilize the output to answer the question.

EVALUATION:

See Figure 7.4.2 for utilization of script output.

SCORES:

Take the direct output as the score.MACOQN10: Are existing module header comment blocks merely copyright info?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py using mode one (1) and utilize the output to answer the question. See Figure 7.4.2 for utilization of script output.

EVALUATION:

Typically, fewer variations in module header size, coupled with a vast majority of the header modules being a single size, indicates that the header modules are just boiler-plate copyright information. The more distributed the sizes of the various module headers are, the more likely it is that header blocks contain additional information. The analysis of the distributions will have to be supplemented by looking at different file's module headers to get a better understanding of their contents. See 7.5.1, 7.5.2, and 7.5.3 for examples on how to interpret the script output for a score.

SCORES:

No	Sizes distributed across many with no consistency
Few	Sizes mostly distributed across many with little consistency
Somewhat	Sizes distributed across many with some consistency, but split
Most	Sizes distributed across few with some consistency
Yes	Almost all size distributed within the same size or sizes

MACOQN11: Estimated percent of data declarations that are commented.

EXEMPLAR TOOL(S):

MaintainabilityComments.py and Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Run the script MaintainabilityComments.py in mode three (3) and utilize the output to answer the question. This output gives a total number of data declarations within the code base. Static analysis tools can then provide additional information on how many declarations are missing comments (e.g., Understand CodeCheck RECOMMEND_16 or CPP_V010).

EVALUATION:

See Figure 7.4.3 for utilization of script/tool output. Doing the following computation then gives the answer to this question:

$$\frac{TOTAL_VARIABLE_DECLARATIONS - NUMBER_OF_UNCOMMENTED_DECLARATIONS}{TOTAL_VARIABLE_DECLARATIONS} \times 100\%$$

SCORES:

Take the direct output as the score.

MACOQN12: Estimated percent of class, function, and method declarations that are commented.

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode four (4) and utilize the output to answer the question.

EVALUATION:

See Figure 7.4.4 for utilization of script output.

SCORES:

Take the direct output as the score.

MACOQN13: Prevalence of commented-out code (which count as comment lines)

EXEMPLAR TOOL(S):

N/A.

PROCESS:

This number is a qualification of MACOQN06.

EVALUATION:

How much commented out code exists will give an idea of prevalence. You can do this by running the following calculation:

$$\frac{COMMENED_OUT_LINES_OF_CODE}{SLOC_COUNT} \times 100\%$$

SCORES:

None	0%
Some	<0.5%
Many	>0.5%

$$\frac{TOTAL_COMMENT_LINES - COMMENTED_OUT_LINES_OF_CODE - TOTAL_MODULE_BLOCK_LINES}{PSLOC} \times 100\%$$

COQN014: Overall the quantity of code comments is:

EXEMPLAR TOOL(S):

N/A.

PROCESS:

This question is a qualification of all the other answers provided in the Comments Quality section.

EVALUATION:

When answering this question, consider the percentage of constructs containing comments, the variability and content of the module header blocks, and the comment-to-code ratio.

SCORES:

Too-few	Percentages are all low (< 50%)
Inconsistent	Percentages are inconsistent (some higher than others)
Appropriate	Percentages are all high (>50%)

7.3 Quality

MACOQL01: Estimated percent of source files with a meaningful preamble comment block re: purpose

EXEMPLAR TOOL(S):

MaintainabilityComments.py

PROCESS:

Interpretation of MACOQN07/09/10 above. This is meant to be an estimation of the total number of files that would contain a meaningful module header. This can be estimated by looking at the output of script MaintainabilityComments.py and looking at the overall distribution of header sizes, and approximating a percentage based on the number of files that have the exact same size as opposed to different sizes. In addition, this will require a sampling of a subset of files' headers to also get a percentage of files without a header block.

EVALUATION:

In general, if there are large groups of files that all have the same length, then those are indicative of boiler plate, and the percentage would include all files but those. As some examples (all of which would need to be further validated by some manual inspection), the rough percentages that would be input for 7.5.1, 7.5.2, and 7.5.3 would be 1.5%, 56.5%, and 100% respectively.

SCORES:

Take the direct output as the score.

MACOQL02: Are code modules separated into standardized sections via comments?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode two (2) and utilize the output to answer the question (the “STANDARDIZED CODE MODULE KEYWORD” section). See 7.4.5 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction, determine whether these are true or false positives.

SCORES:

No	All false positives
Few	<25% true positives
Some	<75% true positives
Many	<100% true positives
Yes	All true positives

MACOQL03: Are C-style /* ... */ comments used?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode one (1) and utilize the output to answer the question. See 7.4.2 for utilization of script output.

EVALUATION:

The script reports an answer for MACOQL04 (see below). Taking the MACOQL04 answer, and performing the following calculation will give a percentage of comments that are C-style comments:

$$\frac{COMMENT_LINES_OF_CODE - CPP_SLASH_SLASH_LINES_OF_CODE}{COMMENT_LINES_OF_CODE} \times 100\%$$

SCORES:

No	0%
Few	<25%
Some	<75%
Many	<100%
Yes	All /* */ comments

MACOQL04: Are // comments used?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode one (1) and utilize the output to answer the question. See 7.4.2 for utilization of script output.

EVALUATION:

Performing the following calculation will give a percentage of comments that are “//” comments:

$$\frac{CPP_SLASH_SLASH_LINES_OF_CODE}{COMMENT_LINES_OF_CODE} \times 100\%$$

SCORES:

No	0%
Few	<25%
Some	<75%
Many	<100%
Yes	All // comments

MACOQL05: Are end-of-line comments used?**EXEMPLAR TOOL(S):**

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode one (1) and utilize the output to answer the question. See 7.4.2 for utilization of script output.

EVALUATION:

Performing the following calculation will give a percentage of lines that contain end-of-line comments:

$$\frac{NUMBER_OF_END_OF_LINE_COMMENTS}{PHYSICAL_LINES_OF_CODE} \times 100\%$$

SCORES:

No	0%
Few	<25%
Some	<75%
Many	<100%
Yes	All lines have end of line comments

MACOQL06: Are block comments spanning multiple lines used?**EXEMPLAR TOOL(S):**

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode one (1) and utilize the output to answer the question. See 7.4.2 for utilization of script output.

EVALUATION:

Performing the following calculation will give a percentage of lines that block comments spanning multiple lines:

$$\frac{NUMBER_OF_BLOCK_COMMENTS_SPANNING_MULTIPLE_LINES}{COMMENT_LINES_OF_CODE} \times 100\%$$

SCORES:

No	0%
Few	<25%
Some	<75%
Many	<100%
Yes	All comment blocks are spanning multiple lines

MACOQL07: Are comments written at the level of the code's intent, e.g. what and why, not how?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode five (5) and utilize the output to answer the question following the sampling process in the introduction. See 7.4.6 for utilization of script output.

EVALUATION:

Analyzing the comments will give a sense for whether the comments are addressing the what and why, and not just the how.

SCORES:

No	Comments do not address what and why at all (or are non-existent)
Few	Few comments address what and why at all (or there are few comments)
Some	Some comments address what and why
Many	Most of the comments address what and why
Yes	There are a good number of comments that address what and why

.

MACOQL08: Are comments used to mark requirements being satisfied in code?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode two (2) and utilize the output to answer the question (the “REQUIREMENTS AND DESIGN KEYWORD” section). See 7.4.5 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives.

SCORES:

No	All false positives
Few	<25% true positives
Some	<75% true positives

Many	<100% true positives
Yes	All true positives

MACOQL09: Are comments used to mark developer changes or modification dates?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode two (2) and utilize the output to answer the question (the “DEVELOPER CHANGES AND MOD KEYWORD” section). See 7.4.5 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives.

SCORES:

No	All false positives
Few	<25% true positives
Some	<75% true positives
Many	<100% true positives
Yes	All true positives

MACOQL10: Do comments exist that relate to constructs from an existing SAS or SDS?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode two (2) and utilize the output to answer the question (the “REQUIREMENTS AND DESIGN KEYWORD” section). See 7.4.5 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives.

SCORES:

No	All false positives
Few	<25% true positives
Some	<75% true positives
Many	<100% true positives
Yes	All true positives

MACOQL11: Number of lines containing suspicious comments (todo, hack, fixme, kludge, later, bug, etc.)

EXEMPLAR TOOL(S):

MaintainabilityComments.py

PROCESS:

Run the script `MaintainabilityComments.py` in mode two (2) and utilize the output to answer the question (the “SUSPICIOUS KEYWORD” section). See 7.4.5 for utilization of script output. The script looks for those keywords listed under `KeywordsList.py` and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives. Based on the analysis of those subset of instances, apply the rough percentage of true positives found to the total number of instances of that keyword, and add all the keyword’s final true positives together to get a final estimate on the number of suspicious keywords used (i.e., if the subset analyzed are all false positives, then don’t count that keyword towards the sum, if the subset analyzed are all true positives, then count the entire keyword count towards the sum, if the subset analyzed are 50% false positives, then count 50% of the total instances towards the sum, etc.).

SCORES:

Take the direct output as the score.

MACOQL12: Estimated percent of comments that are extraction formatted (Doxygen, JavaDoc, etc.)

EXEMPLAR TOOL(S):

`MaintainabilityComments.py`

PROCESS:

Run the script `MaintainabilityComments.py` in mode two (2) and utilize the output to answer the question (the “EXTERNAL TOOL TAG KEYWORD” section). See 7.4.5 for utilization of script output. The script looks for those keywords listed under `KeywordsList.py` and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives. Based on the analysis of those subset of instances, apply the rough percentage of true positives found to the total number of instances of that keyword, and add all the keyword’s final true positives together to get a final estimate on the number of external tag keywords used (i.e., if the subset analyzed are all false positives, then don’t count that keyword towards the sum, if the subset analyzed are all true positives, then count the entire keyword count towards the sum, if the subset analyzed are 50% false positives, then count 50% of the total instances towards the sum, etc.). From there, take that number and divide it by the comment lines of code to get the percentage.

SCORES:

Take the direct output as the score.

MACOQL13: Are extraction formatted (Doxygen, etc.) comments used fully and consistently?

EXEMPLAR TOOL(S):

N/A.

PROCESS:

This question is conditional on MACOQL12.

EVALUATION:

Additionally, as MACOQL12 is addressed by eyes-on-code analysis, the analyst can also get a sense as to whether the tags themselves, if used, are filled out or just exist as blank placeholders. The rough percentage of tags used can also be applied to this question, following the same guiding percentages stated below.

SCORES:

No	<1%
Few	<25% true positives
Some	<75% true positives
Most	<100% true positives
Yes	Used in all files fully and consistently

MACOQL14: Are comments accurate to the code?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode five (5) and utilize the output to answer the question following the sampling process in the introduction. See 7.4.6 for utilization of script output.

EVALUATION:

Analyzing the comments will give a sense for whether the comments are accurate to code.

SCORES:

No	Comments are not accurate to code (or are non-existent)
Few	Few comments are accurate to code (or there are few comments)
Some	Some comments are accurate to code
Most	Most of the comments are accurate to code
Yes	There are enough comments, and they are accurate to code

MACOQL15: Are comments written at a level to be useful for someone unfamiliar with the code?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode five (5) and utilize the output to answer the question following the sampling process in the introduction. See 7.4.6 for utilization of script output.

EVALUATION:

Analyzing the comments will give a sense for whether the comments would be useful to someone unfamiliar with the code.

SCORES:

No	Comments are not useful to someone unfamiliar (or are non-existent)
Few	Few comments are useful (or there are few comments)
Some	Some comments are useful to someone unfamiliar
Most	Most of the comments are useful to someone unfamiliar

Yes	Comments are useful to those unfamiliar with the code
-----	---

MACOQL16: Is the commenting style, level of detail, and formatting consistent throughout?

EXEMPLAR TOOL(S):

N/A.

PROCESS:

Interpretation of all answers provided above.

EVALUATION:

If some comments use multiple lines to describe the purpose and need for the code, while others only describe what the code does, this should adversely impact the score. Also, if there are huge, relatively complex swaths of code, with no comments or the comment formatting is inconsistent, this should negatively influence the score. Additionally, the overall commenting style is inconsistent (e.g., some files have module headers while others do not, a mix of both `/* */` and `//` commenting styles, a mix of using end-of-line-comments) should also negatively affect the score.

SCORES:

No	Style, detail, and formatting is inconsistent
Somewhat	The consistency varies for style, detail, and formatting
Yes	Style, detail, and formatting is consistent

MACOQL17: Are comments indented the same as the corresponding code?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode one (1) and utilize the output to answer the question. See 7.4.2 for utilization of script output.

EVALUATION:

Performing the following calculation will give a percentage of lines that are indented the same:

$$\frac{COMMENT_LINES_OF_CODE - NUMBER_OF_LINES_NOT_ALIGNED_WITH_NEXT}{COMMENT_LINES_OF_CODE} \times 100\%$$

SCORES:

No	0%
Few	<25%
Some	<75%
Most	<100%
Yes	All code is indented the same

MACOQL18: Are comments describing subsequent code lines preceded with (at least) one blank line?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode one (1) and utilize the output to answer the question. See 7.4.2 for utilization of script output.

EVALUATION:

Performing the following calculation will give a percentage of comments without a preceding blank line:

$$\frac{COMMENT_LINES_OF_CODE - NUMBER_OF_LINES_WITHOUT_A_PRECEDING_BLANK_LINE}{COMMENT_LINES_OF_CODE} \times 100\%$$

SCORES:

No	0%
Few	<25%
Some	<75%
Most	<100%
Yes	All code has a blank line preceding it

MACOQL19: Do comments add to or detract from the code's overall understandability?

EXEMPLAR TOOL(S):

MaintainabilityComments.py.

PROCESS:

Run the script MaintainabilityComments.py in mode five (5) and utilize the output to answer the question following the sampling process in the introduction. See 7.4.6 for utilization of script output.

EVALUATION:

Analyzing the comments with give a sense for whether the comments add to the understandability.

SCORES:

Detract	Comments hurt understandability (or there are no comments)
Neither	Comments don't help or hurt understandability (not enough comments)
Add	Comments help understandability of code

MACOQL20: Does the commenting style appear easy to maintain?

EXEMPLAR TOOL(S):

N/A.

PROCESS:

Interpretation of all answers provided above.

EVALUATION:

Significant variations in the style of comments negatively impacts the maintainability. Style includes things like mixing different types of comments, having some in line comments, or not having comments aligned with the code its describing.

SCORES:

No	Style is inconsistent
----	-----------------------

Somewhat	Style is consistent across parts of the code, but not the entire system
Yes	Style is consistent

MACOQL21: Is any portion of the code auto-generated via a code generation tool during the build process?

EXEMPLAR TOOL(S):

Build Tools (native to code's build environment).

PROCESS:

This is a 2-step process. First is to look at any build scripts to see if any file generation is happening. Second, it is possible that file generation happens in a non-obvious manner. Analyze the source directory by looking for any files with timestamps that are marked after the build was created.

EVALUATION:

Look for new or updated files in the source directory. In most cases, at least one build directory is created. By looking for source files (.c, .cpp, .h, etc.), you can see if any files were created in this directory.

SCORES:

No	There are no auto-generated files
Yes	There are autogenerated files

MACOQL22: Is auto-generated code during the build process consistently identifiable via a comment signature?

EXEMPLAR TOOL(S):

Build Tools (native to code's build environment).

PROCESS:

This question is conditional on MACOQL21 and whether any files were found to be auto generated.

EVALUATION:

Visually review ~10% of the files that are autogenerated to see if there are any markings indicating as such in the comment signature.

SCORES:

No	Autogenerated files do not have a comment signature
Yes	Autogenerated files have a comment signature

MACOQL23: Is any portion of the code auto-generated via a code generation tool prior to building?

EXEMPLAR TOOL(S):

N/A.

PROCESS:

Search for directories that might be designated as being generated out of a model or some code generation tool. Also search the various files for any potential model names/tags, looking for anything that is indicative of being auto generated. Additionally, by looking for potential 3rd party copyrights in file headers, those files might be auto generated out of a tool (especially closer to the hardware/BSP levels).

EVALUATION:

Look for files that are in directories that could have been auto generated, or contain tags/markings/names of auto generation tools. By looking at thesesource files (.c, .cpp, .h, etc.), you can see if any files were auto generated..

SCORES:

No	There are no auto-generated files
Yes	There are autogenerated files

MACOQL24: Is any non-build auto-generated code consistently identifiable via a comment signature?

EXEMPLAR TOOL(S):

N/A

PROCESS:

This question is conditional on MACOQL23and whether any files were found to be auto generated.

EVALUATION:

Visually review ~10% of the files that are autogenerated to see if there are any markings indicating as such in the comment signature.

SCORES:

No	Autogenerated files do not have a comment signature
Yes	Autogenerated files have a comment signature

7.4 Script Outputs

Figure 7.4.1 MaintainabilityComments.py initial option selection screen

```
1 *****
2
3 Please pick one of the following options:
4     1. Get Module Header and Comment Information -> Analyze all files and get the distribution/metrics of file headers and comment make up
5     2. Perform Keyword Search -> Search the code for various sets of keywords
6     3. Get Total Number of Variable Declarations -> Get the total number of global and local (file and function) declarations
7     4. Get Number of Comments before Classes or Functions -> Calculate metrics for classes, functions, and methods and preceeding prologs
8     9. Exit
9
10 *****
11
12 User Entry:
```

Figure 7.4.2 MaintainabilityComments.py Option 1 sample output with corresponding mappings

```

1 *****
2
3 *** FILE HEADER AND MODULE INFORMATION ***
4
5 *****
6
7 Number of standard '//' comments: 0 ← MACOQL03, MACOQL04
8 Number of Single lined C-Style '/* ... */' comments: 0 ← MACOQL06
9 Number of block comments spanning more than one line: 0 ← MACOQL05
10 Number of lines with comments in line with code: 0 ← MACOQL18
11 Number of comments without a preceeding blank line: 0 ← MACOQL17
12 Number of comments not aligned with the next line: 0 ← MACOQL17
13
14 *** HEADER DISTRIBUTION ***
15 HEADER LENGTH : NUMBER OF FILES
16 00          : 0
17 05          : 0
18 06          : 0
19 12          : 0
20 13          : 0
21 14          : 0
22 15          : 0
23 16          : 0
24 17          : 0
25 18          : 0
26 20          : 0
27 23          : 0
28 25          : 0
29 31          : 0
30 34          : 0
31 35          : 0
32 36          : 0
33
34 Total Number of files processed: 0
35 Total Number of Module Header Lines: 0 ← MACOQN07
36 Percent of files with module header block: 0 ← MACOQN09
37 Average Lines Per Block: 0
38
39 *****
40

```

Diagram illustrating the mapping of MACOQL and MACOQN codes to specific data fields in the output:

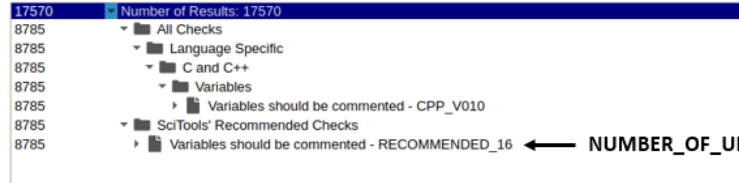
- MACOQL03, MACOQL04: Number of standard '//' comments
- MACOQL06: Number of Single lined C-Style '/* ... */' comments
- MACOQL05: Number of block comments spanning more than one line
- MACOQL18: Number of lines with comments in line with code
- MACOQL17: Number of comments without a preceeding blank line
- MACOQL17: Number of comments not aligned with the next line
- MACOQN10, MACOQL01: Header Length distribution (lines 16-32)
- MACOQN07: Total Number of Module Header Lines
- MACOQN09: Percent of files with module header block

Figure 7.4.3 MaintainabilityComments.py Option 3 sample output with sample static analysis results

```

1 *****
2
3 TOTAL VARIABLES DECLARED AT FILE LEVEL (GLOBALS): 0
4 TOTAL VARIABLES DECLARED AT FILE LEVEL (LOCALS): 0
5 TOTAL VARIABLES DECLARED: 0 ← TOTAL_VARIABLE_DECLARATIONS
6
7 *****

```



17570 Number of Results: 17570

- 8785 All Checks
 - 8785 Language Specific
 - 8785 C and C++
 - 8785 Variables
 - 8785 Variables should be commented - CPP_V010
 - 8785 SciTools' Recommended Checks
 - 8785 Variables should be commented - RECOMMENDED_16 ← NUMBER_OF_UNCOMMENTED_DECLARATIONS

Figure 7.4.4 MaintainabilityComments.py Option 4 sample output with corresponding mappings

```

1 *****
2
3 Number of Functions Processed: 0
4 Number of Functions with Preceeding Comment: 0
5 Percentage of functions that have a perceeding comment: 0
6
7 Number of Classes Processed: 0
8 Number of Classes with Preceeding Comment: 0
9 Percentage of classes that have a perceeding comment: 0
10
11 TOTAL PERCENTAGE OF CLASSES AND FUNCTIONS WITH A PRECEEDING COMMENT: 0 ← MACOQN12
12
13 *****
14

```

Figure 7.4.5 MaintainabilityComments.py Option 2 sample output

```

1  *****
2
3
4  TOTAL NUMBER OF LINES PROCESSED: 0
5  TOTAL NUMBER OF LINES WITH A <CATEGORY> KEYWORD: 0
6  LINES WITH A <CATEGORY> KEYWORD (%): 0
7
8
9  *****
10 ALL <CATEGORY> KEYWORDS AND THE NUMBER OF TIMES EACH APPEAR PER FILE AND WHERE IN EACH FILE
11
12  ✓ KEYWORD #1 [NUMBER_OF_OCCURANCES]
13      FILE_1 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
14      FILE_2 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
15      FILE_3 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
16      ...
17      FILE_N [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
18  ✓ KEYWORD #2 [NUMBER_OF_OCCURANCES]
19      FILE_1 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
20      FILE_2 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
21      FILE_3 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
22      ...
23      FILE_N [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
24
25

```

Figure 7.4.6 MaintainabilityComments.py Option 5 sample output

```

1  *****
2
3  *** NUMBER OF COMMENTS PER FILE DISTRIBUTION ***
4
5  *****
6
7  Number of files processed: 0
8  Number of files listed per distribution: 20 ← N
9
10 -----
11 File                                                                    Comments
12 -----
13 FILE_1_MOST                                                              0
14 FILE_2_MOST                                                              0
15 FILE_3_MOST                                                              0
16 ...
17 FILE_N_MOST                                                              0
18 .....
19 FILE_1_50%                                                                0
20 FILE_2_50%                                                                0
21 FILE_3_50%                                                                0
22 ...
23 FILE_N_50%                                                                0
24 .....
25 FILE_1_LEAST                                                             0
26 FILE_2_LEAST                                                             0
27 FILE_3_LEAST                                                             0
28 ...
29 FILE_N_LEAST                                                             0
30 -----
31 *****
32
33
34
35

```

Visually Inspect the N*3 Files

7.5 Various Scoring Examples

Figure 7.5.1 MACOQN10Example Script Output for score: YES

```
Number of standard '//' comments: 0
Number of Single lined C-Style '/* ... */' comments: 0
Number of block comments spanning more than one line: 0
Number of lines with comments in line with code: 0
Number of comments without a preceeding blank line: 0
Number of comments not aligned with the next line: 0

**** HEADER DISTRIBUTION ****
HEADER LENGTH : NUMBER OF FILES
00           : 10
10           : 2
15           : 1
20           : 1
25           : 2
30           : 1
35           : 1555
40           : 1
45           : 10
50           : 4
55           : 3
```

Because almost all the files are distributed within a single size, this is typically indicative (and validated via inspection) of the header being boiler-plate, standard canned content.

```
Total Number of files processed: 1590
Total Number of Module Header Lines: 55415
Percent of files with module header block: 99.4
Average Lines Per Block: 35.07
```

Figure 7.5.2 MACOQN10Example Script Output for score: SOME

```
Number of standard '//' comments: 0
Number of Single lined C-Style '/* ... */' comments: 0
Number of block comments spanning more than one line: 0
Number of lines with comments in line with code: 0
Number of comments without a preceeding blank line: 0
Number of comments not aligned with the next line: 0

**** HEADER DISTRIBUTION ****
HEADER LENGTH : NUMBER OF FILES
00           : 10
10           : 2
15           : 1
20           : 1
25           : 802
26           : 135
27           : 207
29           : 168
30           : 87
39           : 4
42           : 3
```

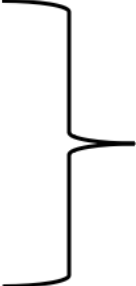
Because most of the files have the exact same header size compared to other sizes, it is assumed (and validated via inspection) that the 802 files' headers are boiler-plate, standard canned content, while the distribution of various other sizes are indicative of there being actual content, (which is also validated by inspection)

```
Total Number of files processed: 1420
Total Number of Module Header Lines: 36968
Percent of files with module header block: 99.3
Average Lines Per Block: 26.2
```

Figure 7.5.3 MACOQN10Example Script Output for score: NO

```
Number of standard '///' comments: 0
Number of Single lined C-Style '/* ... */' comments: 0
Number of block comments spanning more than one line: 0
Number of lines with comments in line with code: 0
Number of comments without a preceeding blank line: 0
Number of comments not aligned with the next line: 0

**** HEADER DISTRIBUTION ****
HEADER LENGTH : NUMBER OF FILES
00           : 10
10           : 2
15           : 1
20           : 1
25           : 202
26           : 135
27           : 207
29           : 168
30           : 87
39           : 4
42           : 3
```



Because not one single size has significantly more files than any other, it is most likely that none of these files are boiler plate canned responses, and contain actual content (which is validated via inspection)

```
Total Number of files processed: 820
Total Number of Module Header Lines: 21968
Percent of files with module header block: 98.7
Average Lines Per Block: 27.1
```

8 Maintainability Clarity

8.1 Overview

Recall, maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [IEEE 1990]. This tab analyzes the clarity of the source code, and whether a lack of understandability increases the overall risk.

To receive a good score in this tab, the code should exhibit a high level of stylistic consistency (naming, source layout, brace usage, etc.), and be carefully crafted throughout the codebase. The code should also frequently use symbolic constants and avoid mixing constants and literals within business logic. The source files and folders should be organized and named to reinforce a coherent architectural understanding. Overall, the lines of code should be easily readable, and the layout, structure, and execution flow should be easily understandable. Lastly, conditional compilation directives should be used sparingly within the code.

For every question, it is recommended analysts investigate files with high cyclomatic complexity (CC), as this can be indicative of a lack of clarity. SA tools provide a variety of functionality that can be used to assist with this section of a CREW analysis. For instance, a treemap visualization shows all the files in the code base and their relative CC. This helps analysts perform a more targeted investigation of files that may be more likely to pose a risk. For most questions, it is recommended that analysts visually review at least ~10% of all the files in the codebase. Analysts should decide upon a set distribution size N (such as 50), and inspect the N files containing the highest or relatively high Cyclomatic complexity (CC), N files close to the 50th percentile in terms of CC, and N files with the lowest or relatively low CC. Some questions in this section of the CREW assessment are designed to make use of common SA tool rule checkers to simplify the analysis process (this is noted in the entry for each question). For these questions, the analyst should visually review at least ~10% of the alerts. Throughout this section, if the term “sample” or “sampling” is used, it refers to the random selection of such a subset of files or alerts for review.

Many questions are answered by selecting a score from a drop-down menu. General guidelines for the approximate thresholds to use when answering each question are provided. However, all assigned scoring must be arbitrated by a human in the loop, and analysts should use discretion when scoring each section. The Maintainability Clarity tab is shown in Figure 8.1.1.

Figure 8.1.1 The Maintainability Clarity Tab

Focus	Risk	W	ID	I Measurement / Question	A Finding
CONSISTENCY	1.3	0.200	MACLCY01	1 Does the overall visual layout clarify the logical structure of the code?	Yes
			MACLCY02	1 Does each line contain only one statement or data declaration?	Yes
			MACLCY03	1 Is source code indented using only space (and not tab) characters?	Yes
			MACLCY04	1 Is the code indentation in consistent integer multiples, e.g. multiples of 3 or 4?	Yes
			MACLCY05	1 Is whitespace used consistently and in a way that clarifies / organizes the code?	Yes
			MACLCY06	1 Estimated total number of commented-out lines of code	• 1035
			MACLCY07	1 Does the name of a type, enum, constant, variable, or routine consistently indicate which one it is?	Yes
			MACLCY08	1 Are variables consistently named to accurately indicate their purpose?	Yes
			MACLCY09	1 Do function or method names indicate precisely what they do?	Yes
			MACLCY10	1 Do functions or methods perform one well-defined purpose?	Yes
			MACLCY11	1 Are braces used consistently to enclose all decision and loop code blocks?	Yes
			MACLCY12	1 Is opening and closing brace placement consistent throughout?	Yes
			MACLCY13	1 Are parentheses used consistently to ensure unambiguous intent?	Yes
CONSTANTS	1.4	0.200	MACLCN01	1 Are well-named constants used instead of magic numbers or magic strings?	Yes
			MACLCN02	1 Estimated number of embedded numeric literals excluding the values (+ and -) 0, 0.0, 1, 1.0, 2, 4, 8, 10, 16	600
			MACLCN03	1 Are constants named for easy identification from variables, e.g. ALL_CAPS?	Yes
			MACLCN04	1 Are there instances of hard-coded paths or filenames within the source code?	No
			MACLCN05	1 Are there instances of hard-coded ports, addresses, or other resources in the source code?	No
			MACLCN06	1 Are global constants defined within a single file or well organized into a few well organized files?	Yes
			MACLCN07	1 Is there evidence that the same global constant is declared/defined in multiple locations?	No
			MACLCN08	1 Are language-standard fixed-width integer types used for integer constant declarations?	Yes
			MACLCN09	1 Are floating point values tested for equality or inequality?	No
			MACLCN10	1 Is const used consistently rather than #define when constants are defined?	Yes
FILES / FOLDERS	1.0	0.100	MACLFF01	1 Is there one class, or collection of routines with one and only one purpose, per file(pair) throughout?	Yes
			MACLFF02	1 Is there one header file, identically named except for the file extension, for each implementation file?	Yes
			MACLFF03	0 Do header files contain #include of other header files?	Yes
			MACLFF04	1 Do subfolder names map 1:1 to documented components or sub-systems?	Yes
			MACLFF05	1 Do header files use preprocessor guards consistently?	Yes
			MACLFF06	1 Does each source file #include the header file of the same root name?	Yes
			MACLFF07	1 Are source files included in other source files?	No
			MACLFF08	0 Does the code utilize any 3rd-party or auto-generated source code?	Yes
READABILITY	3.2	0.300	MACLRF09	1 Is all 3rd-party or auto-generated source code isolated within files and / or sub-folders?	C Yes
			MACLRD01	1 Total number of source files	• 1882
			MACLRD02	0 Number of definition (header) files	• 945
			MACLRD03	0 Number of implementation (code) files	• 937
			MACLRD04	0 Total number of (all) lines in the code base	• 631056
			MACLRD05	1 Source lines of code ignoring blank and comment lines (PSLOC)	• 382397
			MACLRD06	0 Executable source lines of code (ESLOC)	• 38640
			MACLRD07	1 Total number of functions and methods within the code base	• 8157
			MACLRD08	1 Average lines of code per file (PSLOC / total # of source files)	• 203
			MACLRD09	1 Average lines of code per function or method (PSLOC / # functions and methods)	• 47
			MACLRD10	1 Average number of functions or methods implemented per file (# functions and methods / # of impl. files)	• 8.7
			MACLRD11	1 Percentage of functions or methods exceeding 200 lines of code in length	1.0%
			MACLRD12	1 Percentage of source files that exceed 1000 lines of code in length?	2.0%
			MACLRD13	1 Average file cyclomatic complexity (CC1)	• 2.58
			MACLRD14	1 Maximum cyclomatic complexity (CC1) for any function within the code base	• 151
			MACLRD15	1 Percentage of functions or methods within the code exceeding cyclomatic complexity (CC1) of 20	1.0%
			MACLRD16	1 Percentage of functions or methods within the code exceeding strict cyclomatic complexity (CC2) of 20	0.8%
			MACLRD17	1 Maximum control structure nesting level within the code	• 10
			MACLRD18	1 Cyclomatic complexity (total CC1) of the application v(G)	• 17907
			MACLRD19	1 Cyclomatic density (total CC1 / ESLOC)	• 0.46
			MACLRD20	1 Structuredness, i.e. essential complexity (total) of the application Ev(G)	• 9097
			MACLRD21	1 Essential density (total essential complexity / total cyclomatic complexity (CC1))	• 0.51
			MACLRD22	1 Percentage of implementation files with normalized Maintainability Index (MI3) in 'difficult' range (< 10)	15.0%
			MACLRD23	1 Percentage of implementation files with normalized Maintainability Index (MI3) in 'moderate' range (10 >= MI < 20)	45.0%
			MACLRD24	1 Percentage of implementation files with normalized Maintainability Index (MI3) in 'high' range (20 >= MI <= 100)	• 40.0%
			MACLRD25	1 Do source files begin with a meaningful module header block regarding their purpose?	Yes
			MACLRD26	1 Are files separated into consistent sections, e.g. file preamble, includes, defines, data types, functions, etc?	Yes
			MACLRD27	1 Are methods or functions within a file separated with a blank line?	Yes
			MACLRD28	1 Is the nominal path through the code generally clear to the reader?	Yes
			MACLRD29	1 Does the code use simple constructs rather than uncommon language features?	Yes
			MACLRD30	1 Does the code avoid complex multiple conditions within decisions?	Yes
			MACLRD31	1 Do loops and functions have just one exit path?	Yes
			MACLRD32	1 Does the code make use of 'goto' to modify the execution path?	No
			MACLRD33	1 Number of occurrences of direct or indirect recursion within the code	4
			MACLRD34	1 Are functions implementing tasks or threads named in a way to make this clear? E.g. ending in "_task"?	Yes
CONDITIONALS	2.8	0.200	MACLCD01	1 Number of unique conditional compile flags used within the code	50
			MACLCD02	1 Are conditionals prevalent enough within the code to hinder understanding it?	No
			MACLCD03	0 Is there a mechanism to indicate at run-time which conditionals were used at build-time?	No
RISK	2.14				

The following sections describe heuristics for scoring the CREW Maintainability Clarity tab:

8.2 Consistency

MACLCY01: Does the overall visual layout clarify the logical structure of the code?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Review the placement of include statements, data definitions, functions, etc. and verify the structure is logical and well organized by type via visual code inspection (focus on individual files) of

a sampling of files. Also, static analysis tool rules (e.g., Understand Code Check Rule CPP_P017) that flag inconsistent or poorly grouped code sections (e.g. include blocks) can be used to generate candidates.

EVALUATION:

For this question, the perspective is primarily focused on whether individual files are logically organized, and have the same consistent layout. A good convention is to begin a file with a header comment block, followed by a block of include statements, a block of constant variable declarations, perhaps a few function declarations, and then the main body of the code. If any of these sections is missing or mixed with another section, this negatively impacts the score.

SCORES:

No	$\geq 25\%$ of files have mixed or missing sections.
Seldom	$< 25\%$ of files have mixed or missing sections.
Sometimes	$< 10\%$ of files have mixed or missing sections.
Usually	$< 5\%$ of files have mixed or missing sections.
Yes	$< 1\%$ of files have mixed or missing sections.

MACLCY02: Does each line contain only one statement or data declaration?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand Code Check rule A7-1-7) that flag multiple declarations or statements to find candidate lines containing violations.

EVALUATION:

Verify whether each line is a true positive within a sampling of candidates with visual code inspection.

SCORES:

No	≥ 200 total true positives.
Seldom	≥ 100 and < 200 total true positives.
Sometimes	≥ 25 and < 100 total true positives.
Usually	≥ 5 and < 25 total true positives.
Yes	< 5 total true positives.

MACLCY03: Is source code indented using only space (and not tab) characters?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand), and WhitespaceAnalysis.py script.

PROCESS:

Use static analysis tool rules and the WhitespaceAnalysis.py script to compute the number of tabs. See 8.8.1 for an example of script output.

EVALUATION:

The number of tabs detected is used directly to calculate the score.

SCORES:

No	(SLOC) / (# of tabs) < 25
Seldom	(SLOC) / (# of tabs) >= 25 and < 50
Sometimes	(SLOC) / (# of tabs) >= 50 and < 75
Usually	(SLOC) / (# of tabs) >= 75 and < 100
Yes	(SLOC) / (# of tabs) >= 100

MACLCY04: Is the code indentation in consistent integer multiples, e.g., multiples of 3 or 4?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand), and WhitespaceAnalysis.py script.

PROCESS:

Use static analysis tool rules and the WhitespaceAnalysis.py script to find indentation information and verify flagged files are true positives with visual code inspection. See 8.8.1 for an example of script output.

EVALUATION:

The distribution of tabs detected is used directly to calculate the score. A sample of files are visually reviewed to verify the output of the script is in line with expectations.

SCORES:

No	<=50% of files are indented at the same multiple.
Seldom	>50% of files are indented at the same multiple.
Sometimes	>70% of files are indented at the same multiple.
Usually	>90% of files are indented at the same multiple.
Yes	>99% of files are indented at the same multiple.

MACLCY05: Is whitespace used consistently and in a way that clarifies / organizes the code?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand), and WhitespaceAnalysis.py script.

PROCESS:

Use static analysis tool rules and the WhitespaceAnalysis.py script to find indentation information. See 8.8.1 for an example of script output. Also verify a sample of files via visual code inspection.

EVALUATION:

This question focuses on determining if whitespace is used consistently and is less concerned with whether tabs or spaces were used. Repeated instances of poor spacing/indentation (e.g., not using whitespace to convey separate blocks of code), negatively impacts the score.

SCORES:

No	>=25% of files use whitespace poorly.
Seldom	<25% of files use whitespace poorly.
Sometimes	<10% of files use whitespace poorly.

Usually	<5% of files use whitespace poorly.
Yes	<1% of files use whitespace poorly.

MACLCY06: Estimated total number of commented-out lines of code.

From System tab, Metrics Focus, Item #22.

MACLCY07: Does the name of a type, enum, constant, variable, or routine consistently indicate which one it is?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search for common types (e.g., “enum”, “const”, etc.) in a static analysis tool to find candidate files to review with visual code inspection. Then visually review a sample of the candidate files.

EVALUATION:

Determine if variable names tend to indicate their type (e.g., name of a “const” type variable has the prefix “const_”). Repeated instances of names that do not indicate the type can be considered true positives and negatively impact the score.

SCORES:

No	>= 80% of inspected variables, etc. are a true positive.
Seldom	>= 60% and < 80% of inspected variables, etc. are a true positive.
Sometimes	>= 30% and < 60% of inspected variables, etc. are a true positive.
Usually	>= 5% and < 30% of inspected variables, etc. are a true positive.
Yes	< 5% of inspected variables, etc. are a true positive.

MACLCY08: Are variables consistently named to accurately indicate their purpose?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Review the variables identified by the static analysis tool or visually inspect a sampling of files and inspect the naming/structural quality of the variables.

EVALUATION:

Determine if the variables have descriptive names with full words or understandable abbreviations like “LightOn” or “TogSpeaker” instead of single characters or other vague names, which can be thought of as true positives.

SCORES

No	>= 200 total true positives distributed throughout several files of the codebase.
Seldom	>= 100 and < 200 total true positives distributed throughout several files of the codebase.
Sometimes	>= 5 and < 100 total true positives distributed throughout several files of the codebase.
Usually	>= 5 and true positives but localized to a few files of the codebase.
Yes	< 5 total true positives.

MACLCY09: Do function or method names indicate precisely what they do?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Review the functions or methods identified by the static analysis tool. Visually inspect a sampling of files and review the naming/structural quality of the functions or methods.

EVALUATION:

Determine if functions have descriptive names with full words or understandable abbreviations like “SetLightSwitch” or “CreatSpeakerArr” instead of single characters or otherwise vague names, which can be considered true positives.

SCORES:

No	>= 200 total true positives distributed throughout several files of the codebase.
Seldom	>= 100 and < 200 total true positives distributed throughout several files of the codebase.
Sometimes	>= 5 and < 100 total true positives distributed throughout several files of the codebase.
Usually	>= 5 and true positives but localized to a few files of the codebase.
Yes	< 5 total true positives.

MACLCY10: Do functions or methods perform one well-defined purpose?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Look for files with high cyclomatic complexity or long functions and review the naming/structural quality of the functions or methods.

EVALUATION:

Judge whether the functions in a sampling of files are self-contained and easy to follow with minimal background on the requirements of the application. Functions that are very long and/or have many conditional branches may be more likely to not perform one well-defined purpose and can be thought of as a true positive. The more of these functions that are found, the more negative the score will be.

SCORES:

No	>= 200 total true positives.
Seldom	>= 100 and < 200 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 5 and < 25 total true positives.
Yes	< 5 total true positives.

MACLCY11: Are braces used consistently to enclose all decision and loop code blocks?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand Code Check rule CPP_ST021) to generate candidate lines that may lack braces to review.

EVALUATION:

Review a sampling of the lines with visual code inspection to verify brace usage consistency. This question looks to see if braces are used regularly to clarify where decisions and loops are in the code. A lack of braces in a decision or loop block is considered a true positive. The greater the frequency of missing braces around decision and loop blocks the more negative the score.

SCORES:

No	>= 200 total true positives.
Seldom	>= 100 and < 200 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 5 and < 25 total true positives.
Yes	< 5 total true positives.

MACLCY12: Is opening and closing brace placement consistent throughout?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand Code Check rules CPP_ST021 and CPP_ST003) to generate candidate lines to review.

EVALUATION:

Review the lines with visual code inspection to verify brace usage consistency. This question checks to see if braces are regularly placed in the same relative location to their associated statements (e.g., whether opening braces are placed on the same line of an if conditional statement or on the following line). The greater the variation of brace placement the more negative the score.

SCORES:

No	>=25% of files contain a deviation from the established convention.
Seldom	<25% of files contain a deviation from the established convention.

Sometimes	<10% of files contain a deviation from the established convention.
Usually	<5% of files contain a deviation from the established convention.
Yes	<1% of files contain a deviation from the established convention.

MACLCY13: Are parentheses used consistently to ensure unambiguous intent?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand Code Check rules CPP_E042, CPP_PR038, and CPPS_T009) to generate candidate lines to review. Review the lines from a sampling of files with visual code inspection to verify usage consistency.

EVALUATION:

This question ascertains whether parentheses are consistently used in expressions and decisions to clarify the expected order of operations and the grouping of associated operands. A lack of parentheses may lead to ambiguous code that is difficult to understand and may have unintended behavior, especially in macros. Expressions that depend on the language order of operations and macros are not fully parenthesized are considered true positives, and negatively impact the score.

SCORES:

No	≥ 200 total true positives.
Seldom	≥ 100 and < 200 total true positives.
Sometimes	≥ 25 and < 100 total true positives.
Usually	≥ 5 and < 25 total true positives.
Yes	< 5 total true positives.

8.3 Constants

MACLCN01: Are well-named constants used instead of magic numbers or magic strings?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., UnderstandCode Check rule RECOMMENDED_08) to generate candidate lines to review.

EVALUATION:

Check with visual code inspection to verify whether hardcoded values appear frequently in a sampling of the flagged lines. Hardcoded numerical constants can obscure the intentions of the author of the code and make a program more challenging to maintain and extend. Hardcoded constants in the code are considered true positives, and negatively impact the score.

SCORES:

No	≥ 200 total true positives, and little to no evidence of the use of well-named constants.
Seldom	≥ 200 total true positives.

Sometimes	>= 100 and < 200 total true positives.
Usually	>= 25 and < 100 total true positives.
Yes	< 25 total true positives.

MACLCN02: Estimated number of embedded numeric literals excluding the values (+ and -) 0, 0.0, 1, 1.0, 2, 4, 8, 10, 16.

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., UnderstandCode Check rule RECOMMENDED_08) to generate candidate lines to review.

EVALUATION:

Review a sampling of the lines with visual code inspection to verify if hardcoded values are present. Based on the percentage of true positives among the sample of values inspected, estimate the number of true positives contained in all the candidate lines and report this estimate. For example, if 10,000 candidate lines are flagged, and 100% of the sample is true positives, report 10,000 as the estimate.

SCORES:

N/A

MACLCN03: Are constants named for easy identification from variables, e.g., ALL_CAPS?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search inside the codebase for static/const variables using regex 'static\sconst\s' and its variations, and then verify a sampling of results with visual code inspection.

EVALUATION:

If constants are not well named compared to other variables (e.g., prefixed with 'c', 'g', ALL_CAPS, etc.), this should be noted as a true positive and negatively impacts the score.

SCORES:

No	>= 200 total true positives.
Seldom	>= 100 and < 200 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 5 and < 25 total true positives.
Yes	< 5 total true positives.

MACLCN04: Are there instances of hard-coded paths or filenames within the source code?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand) and/or KeywordSearch.py script.

PROCESS:

Search inside the codebase using regex `'(\\|\\).*(\\|\\).*'` or for file extensions to look for candidate lines of code that contain filenames or hardcoded paths and verify a sampling of results with visual code inspection.

EVALUATION:

If a result is found to be a true positive, select yes. This negatively impacts the score.

SCORES:

No	No true positives.
Yes	At least one true positive.

MACLCN05: Are there instances of hard-coded ports, addresses, or other resources in the source code?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search inside the codebase using the regex `'(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?).(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?).(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?).(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)'` from <https://regex101.com/r/dT0vT3/1>, and `'(")([1-9][0-9]{0,3}[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5])(")'` from <https://stackoverflow.com/questions/12968093/regex-to-validate-port-number> and then verify a sampling of results with visual code inspection.

EVALUATION:

If a result is found to be a true positive, select yes. This negatively impacts the score.

SCORES:

No	No true positives.
Yes	At least one true positive.

MACLCN06: Are global constants defined within a single file or well organized into a few files?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tools to generate candidate lines and check with visual code inspection to verify if global constants are defined within a single file or well organized into a few files based on

the library structure, namespace, usage context, etc.. Also, search in the codebase for related global constant keywords (e.g., extern) and verify with visual code inspection.

EVALUATION:

Determine whether a sampling of the results in different files are true positives and answer appropriately.

SCORES:

No	Global constants in different, disorganized files.
Few	A minority of the global constants are in the same file or well organized into a few files.
Some	About half of the global constants are in the same file or well organized into a few files.
Many	A majority of the global constants are in the same file or well organized into a few files.
Yes	All global constants in the same file or well organized into a few files.

MACLCN07: Is there evidence that the same global constant is declared/defined in multiple locations?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tools to generate candidate lines and check with visual code inspection to verify. Also, search in the codebase for related global constant keywords (e.g., extern) and verify with visual code inspection.

EVALUATION:

Determine whether a sampling of the results in different files are true positives and answer appropriately.

SCORES:

No	No global constants declared/defined more than once.
Some	A few global constants declared/defined more than once.
Yes	Global constants declared/defined in different files.

MACLCN08: Are language-standard fixed-width integer types used for integer constant declarations?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Compare regex searching for

```
'const.*(int8_t|int16_t|int32_t|int64_t|int_fast8_t|int_fast16_t|int_fast32_t|int_fast64_t|int_least8_t|
int_least16_t|int_least32_t|int_least64_t|intptr_t|uint8_t|uint16_t|uint32_t|uint64_t|uint_fast8_t|uin
t_fast16_t|uint_fast32_t|uint_fast64_t|uint_least8_t|uint_least16_t|uint_least32_t|uint_least64_t|uin
```

tmax_t|uintptr_t|intmax_t)' and 'const.*(int|short)'. Then repeat and include the keyword 'static' in the regex.

EVALUATION:

Analyze the differences between a sampling of the results to determine the score. Instances where fixed width types could have been used but were not utilized are considered true positives. The more prevalent these instances are, the worse the score.

SCORES:

No	>= 200 total true positives, and no evidence of any fixed width type usage.
Seldom	>= 200 total true positives.
Sometimes	>= 100 and < 200 total true positives.
Usually	>= 25 and < 100 total true positives.
Yes	< 25 total true positives.

MACLCN09: Are floating point values tested for equality or inequality?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand Code Check rule RECOMMENDED_03) to generate candidate lines and verify with visual code inspection.

EVALUATION:

If any result is a true positive, mark the score as appropriate.

SCORES:

No	No true positives.
Seldom	< 25 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 100 and < 200 total true positives.
Yes	>= 200 total true positives.

MACLCN10: Is const used consistently rather than #define when constants are defined?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search in the codebase for the regex 'define\s+.\s+.\s+.' to generate candidate lines and verify with visual code inspection. Static analysis tools can also be used to generate candidate lines.

EVALUATION:

Identifying an instance where define is used instead of const is considered a true positive and hurts the score.

SCORES:

No	>= 100 total true positives and no use of const detected.
Seldom	>= 100 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 5 and < 25 total true positives.
Yes	< 5 total true positives.

8.4 Files / Folders

MACLFF01: Is there one class, or collection of routines with one and only one purpose, per file(pair) throughout?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search for 'class' to generate candidate files and verify a sampling of results with visual code inspection. Also, review a sample of the files to look for collections of routines and determine whether they have a unified purpose.

EVALUATION:

If instances where a class is present in a file already containing a class, or a collection of routines is found without a unified purpose, this is considered a true positive and negatively impacts the score. Whether or not the true positives are localized to a few files or distributed throughout the codebase may influence the score, especially at threshold boundaries.

SCORES:

No	>= 200 total true positives and/or distributed throughout the codebase.
Seldom	>= 100 and < 200 total true positives and/or distributed throughout the codebase.
Sometimes	>= 5 and < 100 total true positives and/or distributed throughout the codebase.
Usually	>= 5 total true positives localized to a few files in the codebase.
Yes	< 5 total true positives localized to a few files in the codebase.

MACLFF02: Is there one header file, identically named except for the file extension, for each implementation file?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Select a sampling of files and verify whether the file is not part of a header-source pair as the question describes via visual code inspection of the directory of files comprising the codebase.

EVALUATION:

Source files that are not part of a header-source pair are considered true positives and negatively impact the score.

SCORES:

No	>=50% of files are true positives.
Seldom	< 50% of files are true positives.
Sometimes	< 25% of files are true positives.
Usually	< 10% of files are true positives.
Yes	< 1% of files are true positives.

MACLFF03: Do header files contain #include of other header files?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Select a sampling of files and verify whether the files include other headers as the question describes with visual code inspection.

EVALUATION:

Header files that include other header files are considered true positives and negatively impact the score.

SCORES:

No	< 1% of files are true positives.
Few	< 10% of files are true positives.
Some	< 25% of files are true positives.
Many	< 50% of files are true positives.
Yes	>=50% of files are true positives.

MACLFF04: Do subfolder names map 1:1 to documented components or sub-systems?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Review the documentation combined with the directory structure to evaluate how well the project maps to the documentation.

EVALUATION:

Increasing numbers of subfolders that do not map to documented components hurt the score.

SCORES:

No	Not clear from inspection, no documentation, or outside help needed.
Few	A few folders map to documented components.
Some	~50% of folders map to documented components.
Many	Most folders map to documented components.
Yes	All or almost all folders map to documented components.

MACLFF05: Do header files use preprocessor guards consistently?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand CodeCheck rule MISRA-C 2004) to generate candidate lines and verify with visual code inspection. Also, review a sample of the files in the code-base and make sure header guards are used for each file. Note where discrepancies occur.

EVALUATION:

Files without header guards are considered true positives and negatively impact the score.

SCORES:

No	>= 25% of files are a true positive.
Few	< 25% of files are a true positive.
Some	< 10% of files are a true positive.
Many	< 5% of files are a true positive.
Yes	< 1% of files are a true positive.

MACLFF06: Does each source file #include the header file of the same root name?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Select a sampling of source files and verify whether they #include the header file of the same root name as the question describes (excluding common exceptions, such as unit tests and small source files containing just a main() function) with visual code inspection.

EVALUATION:

An increasing number of true positives (source files that do not #include the header file of the same root name) negatively impacts the score.

SCORES:

No	>=50% of files are true positives.
Few	< 50% of files are true positives.
Some	< 25% of files are true positives.
Many	< 10% of files are true positives.
Yes	< 1% of files are true positives.

MACLFF07: Are source files included in other source files?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Select a sampling of source files and verify whether they #include source files as the question describes with visual code inspection.

EVALUATION:

If any result is a true positive, mark the score as appropriate .

SCORES:

No	0 true positives.
Few	<2 true positives.
Some	<3 true positives.
Many	<5 true positives.
Yes	>=5 true positives.

MACLFF08: Does the code utilize any 3rd-party or auto-generated source code?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search in the codebase for ‘copyright’, ‘autogen’, etc. to generate candidate lines or files containing 3rd-party or auto-generated source code and verify the results with visual code inspection.

EVALUATION:

Select yes if any files from a third party or autogenerated code is found.

SCORES:

No	No true positives.
Yes	At least one true positive.

MACLFF09: Is all 3rd-party or auto-generated source code isolated within files and / or sub-folders?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search in the codebase for ‘copyright’, ‘autogen’, etc. for candidate lines of 3rd-party or auto-generated source code and verify with visual code inspection. Compare the findings against the directory structure of the project to ascertain whether 3rd party code is primarily located in one location or spread throughout the codebase.

EVALUATION:

The more 3rd-party or auto-generated code is spread throughout the codebase or files, the worse the score.

SCORES:

No	There is no noticeable standard or convention for organizing third party software in the codebase.
Seldom	Most third-party software is spread throughout the codebase.

Sometimes	Several files from third parties are mixed with software from the primary organization in the same folder, or third-party software is distributed across folders and is not centrally located.
Usually	A few files from third parties are mixed with software from the primary organization in the same folder.
Yes	All third-party software is isolated in distinct folders.

8.5 Readability

MACLRD01: Total number of source files

From System tab, Metrics Focus, Item #8.

MACLRD02: Number of definition (header) files

From System tab, Metrics Focus, Item #9.

MACLRD03: Number of implementation (code) files

From System tab, Metrics Focus, Item #10.

MACLRD04: Total number of (all) lines in the code base

From System tab, Metrics Focus, Item #11.

MACLRD05: Source lines of code ignoring blank and comment lines (PSLOC)

From System tab, Metrics Focus, Item #12.

MACLRD06: Executable source lines of code (ESLOC)

From System tab, Metrics Focus, Item #13.

MACLRD07: Total number of functions and methods within the code base

From System tab, Metrics Focus, Item #16.

MACLRD08: Average lines of code per file (PSLOC / total # of source files)

Automatically computed from the metrics listed in MACLRD05 and MACLRD01.

MACLRD09: Average lines of code per function or method (PSLOC / # functions and methods)

Automatically computed from the metrics listed in MACLRD05 and MACLRD07.

MACLRD10: Average number of functions or methods implemented per file (# functions and methods / # of impl. Files)

Automatically computed from the metrics listed in MACLRD07 and MACLRD03.

MACLRD11: Percentage of functions or methods exceeding 200 lines of code in length.

Compute with static analysis tool rule (e.g., Understand Code Check rule RECOMMENDED_04) and the metric in MACLRD07. Note the equation to compute this is

$$\frac{(\text{NUMBER_OF_FUNCTIONS_OR_METHODS_EXCEEDING_200_LINES})}{(\text{TOTAL_NUMBER_OF_FUNCTIONS_OR_METHODS})}$$

MACLRD12: Percentage of source files that exceed 1000 lines of code in length?

Compute with static analysis tool rule (e.g., Understand Code Check rule RECOMMENDED_02) and the metric in MACLRD01. Note the equation to compute this is

$$\frac{(\text{NUMBER_OF_SOURCE_FILES_EXCEEDING_1000_LINES})}{(\text{TOTAL_NUMBER_OF_SOURCE_FILES})}$$

MACLRD13: Average file cyclomatic complexity (CC1)

From System tab, Metrics Focus, Item #17.

MACLRD14: Maximum cyclomatic complexity (CC1) for any function within the code base

From System tab, Metrics Focus, Item #18.

MACLRD15: Percentage of functions or methods within the code exceeding cyclomatic complexity (CC1) of 20

Compute static analysis tool rule (e.g., Understand Code Check rule RECOMMENDED_10) and the metric in MACLRD07.

MACLRD16: Percentage of functions or methods within the code exceeding strict cyclomatic complexity (CC2) of 20

Compute with static analysis tool rule (e.g., Understand Code Check rule RECOMMENDED_10) and the metric in MACLRD07.

MACLRD17: Maximum control structure nesting level within the code

From System tab, Metrics Focus, Item #19.

MACLRD18: Cyclomatic complexity (total CC1) of the application v(G)

From System tab, Metrics Focus, Item #20.

MACLRD19: Cyclomatic density (total CC1 / ESLOC)

Automatically computed from System tab, Metrics Focus, Item #17 and the metric in MACLRD18.

MACLRD20: Structuredness, e.g. essential complexity (total) of the application Ev(G)

From System tab, Metrics Focus, Item #21.

MACLRD21: Essential density (total essential complexity / total cyclomatic complexity (CC1))

Automatically computed from the metrics from MACLRD20 and MACLRD18.

MACLRD22: Percentage of implementation files with normalized Maintainability Index (MI3) in 'difficult' range (< 10)

Compute from static analysis tool rule (e.g., Understand Code Check rule METRIC_13).

MACLRD23: Percentage of implementation files with normalized Maintainability Index (MI3) in 'moderate' range (≥ 10 MI < 20)

Compute from static analysis tool rule (e.g., Understand Code Check rule METRIC_13).

MACLRD24: Percentage of implementation files with normalized Maintainability Index (MI3) in 'high' range (≥ 20)

Automatically computed in the CREW form by computing the formula $(1 - (\text{MACLRD22} - \text{MACLRD23}))$.

MACLRD25: Do source files begin with a meaningful module header block regarding their purpose?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Select a sample of source files and leverage visual code inspection to review whether header blocks contain useful information. Make sure that the header block is more than just boilerplate and describes information including but not necessarily limited to the important dates in the history of the file, the authors, the purpose of the code in the file, and/or relevant maintainability related information.

EVALUATION:

If source files lack a header entirely or only contain a boilerplate header with minimal information (e.g., licensing information) this can be thought of as a true positive and negatively impacts the score.

SCORES:

No	The vast majority of the files do not have a useful header block.
Seldom	At least a few of the files have a useful header block.
Sometimes	Approximately half of the files have a useful header block.
Usually	Most of the files have a useful header block.
Yes	All or almost all the files have a useful header block.

MACLRD26: Are files separated into consistent sections, e.g., file preamble, includes, defines, data types, functions, etc.?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Select a sample of files and review the placement of include statements, data definitions, functions, etc. to verify the structure is logical and well organized by type via visual code inspection (focus on individual files). Also, static analysis tools can be used to generate candidate files to inspect.

EVALUATION:

Mixing statements outside of standard sections, such as having an “include” statement in the middle of the main body of a source file, or a noticeably disorganized file can be thought of as a true positive and negatively impacts the score.

SCORES:

No	>= 200 total true positives.
Seldom	>= 100 and < 200 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 5 and < 25 total true positives.
Yes	< 5 total true positives.

MACLRD27: Are methods or functions within a file separated with a blank line?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use static analysis tool rules (e.g., Understand Code Check rule CPP_ST024) to generate candidate functions that lack spaces for separation and analyze a sample of the lines with visual code inspection to determine whether they are true positives.

EVALUATION:

Increasing numbers of true positives negatively impacts the score.

SCORES:

No	>= 200 total true positives.
Seldom	>= 100 and < 200 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	>= 5 and < 25 total true positives.
Yes	<5 total true positives.

MACLRD28: Is the nominal path through the code generally clear to the reader?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search for “continue” and “goto” statements and/or functions with multiple return paths. Many of these statements may make the path through the code hard to follow. Evaluate whether conditional compilation impacts the ability to find the active code. Also review files for general clarity and understandability.

EVALUATION:

“continue” and “goto” statements and/or functions with multiple return paths may be thought of as true positives depending on the context and suggests to the analyst to enter a negative score.

Also, if reading through a subsection of the code to determine the execution path of the program presents challenges, this should be reflected in the final score for the question. Ultimately this question should reflect the analyst's experience reading through the code, and not be solely informed by the frequency of specific code constructs.

SCORES:

No	The code is very hard to follow (e.g., ≥ 200 total true positives).
Seldom	There are many confusing sections (e.g., ≥ 100 and < 200 total true positives).
Sometimes	There are several confusing sections (e.g., ≥ 25 and < 100 total true positives).
Usually	There are a few confusing sections (e.g., ≥ 5 and < 25 total true positives).
Yes	The path is clear (e.g., < 5 total true positives).

MACLRD29: Does the code use simple constructs rather than uncommon language features?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand). KeywordSearch.py.

PROCESS:

Select a sample of files to investigate and leverage visual code inspection to detect obscure feature use. If the use of a search engine is frequently required to understand what a section of code is doing, this is indicative of a lack of simple constructs. What is considered uncommon may vary considerably between software engineers, and it is recommended that a coding standard be agreed upon to use when deciding what features to regard as uncommon, such as any features added following the release of C++17. Alternatively, a specific list of features considered uncommon by the CREW analyst (such as use of the 'virtual' keyword, etc.) could be used when answering this question (to enable searching for keywords or coding constructs within the codebase that are considered evidence of uncommon feature use).

EVALUATION:

Each instance of the use of an uncommon or obscure language feature can be thought of as a true positive, and negatively impacts the score.

SCORES:

No	$\geq 25\%$ of files have a true positive.
Seldom	$< 25\%$ of files have a true.
Sometimes	$< 10\%$ of files have a true positive.
Usually	$< 5\%$ of files have a true positive.
Yes	No or almost no files have a true positive.

MACLRD30: Does the code avoid complex multiple conditions within decisions?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search in a static analysis tool for multiple conditions using variations of the regex “if\s*\(.*(\&\&|\||\)|)” and verify the results with visual code inspection. If there are many conditions, this may indicate a lack of clarity.

EVALUATION:

Conditional statements with more than four conditions can be thought of as true positives, and negatively impact the score.

SCORES:

No	>= 200 total true positives.
Seldom	>= 100 and < 200 total true positives.
Sometimes	>= 25 and < 100 total true positives.
Usually	< 25 total true positives.
Yes	< 5 total true positives.

MACLRD31: Do loops and functions have just one exit path?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Run static analysis tool rules (e.g., Understand Code Check rule CPP_C029) to generate candidates that have multiple exit paths and verify a sampling of the results with visual code inspection to determine if these are true/false positives. Also, review the functions in a random sampling of files and determine whether the functions have just one exit path.

EVALUATION:

Each function with more than one exit path can be thought of as a true positive, and increasing numbers of these negatively impacts the score.

SCORES:

No	>= 80% of functions are a true positive.
Seldom	>= 60% and < 80% of functions are a true positive.
Sometimes	>= 30% and < 60% of functions are a true positive.
Usually	>= 5% and < 30% of functions are a true positive.
Yes	< 5% of functions are a true positive.

MACLRD32: Does the code make use of ‘goto’ to modify the execution path?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Run static analysis tool rules (e.g., Understand Code Check rule CPP_C025) to generate candidates and verify if the results are true positives with visual code inspection.

EVALUATION:

Each ‘goto’ statement counts as a true positive and increasing numbers of true positives negatively impacts the score.

SCORES:

No	0 true positives.
Few	<2 true positives.
Some	<3 true positives.
Many	<5 true positives.
Yes	>=5 true positives.

MACLRD33: Number of occurrences of direct or indirect recursion within the code

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Run static analysis tool rules (e.g., Understand Code Check rule AC_01) to generate candidates and verify if the results or a sampling of the results are true positives with visual code inspection. For example, if 10,000 candidate lines are flagged, and 100% of the samples are true positives, report 10,000 as the estimate.

EVALUATION:

Directly input the numerical estimate into the CREW spreadsheet.

SCORES:

N/A

MACLRD34: Are functions implementing tasks or threads named in a way to make this clear? E.g., ending in "_task"?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search for ‘thread’, ‘task’, and related keywords in a sample of files in the codebase to generate candidates and then visually inspect the code to verify the results’ clarity. Searching for the use of common threading or multiprocessing libraries (e.g. pthreads) may help answer this question.

EVALUATION:

A true positive is a function implementing tasks or threads that is not clearly named to indicate its purpose. The more true positives, the worse the score.

SCORES:

No	>= 25% of functions are true positives.
Seldom	< 25% of functions are true positives.
Sometimes	< 10% of functions are true positives.
Usually	< 5% of functions are true positives.
Yes	No or almost no true positives.

8.6 Conditionals

MACLCD01: Number of unique conditional compile flags used within the code.

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use the ConditionalCompiles.py script to generate candidates, and then verify the results or a sample of the results with visual code inspection. See 8.8.2 for example of script output. For example, if 10,000 candidate flags are found, and 100% of the samples are true positives, report 10,000 as the estimate.

EVALUATION:

The script can be used to compute a numerical estimate to enter in the spreadsheet, although this value should be verified with visual code inspection of the results or a sample of the results.

MACLCD02: Are conditionals prevalent enough within the code to hinder understanding it?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Use the ConditionalCompiles.py script to generate candidates, and then verify the flagged files contain true positives with visual code inspection. See 8.8.2 for example of script output. For this question, be sure to look for files with multiple conditionals.

EVALUATION:

Inspect how intuitive the path through the compilation process is for new maintainers who did not author the code. Code that is challenging to grasp without resorting to using a search engine or spending more than several minutes reading through the code negatively impacts the score.

SCORES:

No	Conditionals are generally straightforward.
Somewhat	A few conditionals are difficult to understand.
Yes	Many conditionals are difficult to understand.

MACLCD03: Is there a mechanism to indicate at run-time which conditionals were used at build-time?

EXEMPLAR TOOL(S):

Static analysis tools (e.g., SciTools Understand).

PROCESS:

Search for “runtime”, “conditional” and related keywords and use visual code inspection to look and see if such a mechanism exists (especially files flagged via the ConditionalCompiles.py script) in a sample of files.

EVALUATION:

Mark this question appropriately depending on whether such a mechanism was found.

SCORES:

No	The mechanism was not found.
Yes	The mechanism was found.

8.7 Script Outputs

Figure 8.8.1 WhitespaceAnalysis.py sample output

```
TOTAL NUMBER OF FILES PROCESSED: 0

TOTAL TABS USED IN THE FILE: 0 ← MACLCY03

WHITESPACE INTERVALS FOR OVERALL CODE BASE ← MACLCY04
INTERVAL | PERCENTAGE
-----
3      | 00.00
4      | 00.00
5      | 00.00

*****

WHITESPACE INTERVALS PER FILE ← MACLCY05
/file/path/example
INTERVAL | PERCENTAGE
-----
3      | 00.00
4      | 00.00
5      | 00.00
...

*****
ALL FILES (WITH LINE NUMBERS) THAT CONTAIN AT LEAST ONE TAB

/file/path/example: LINE1, LINE2, ...
/file/path/example2: LINE1, LINE2, ...
```

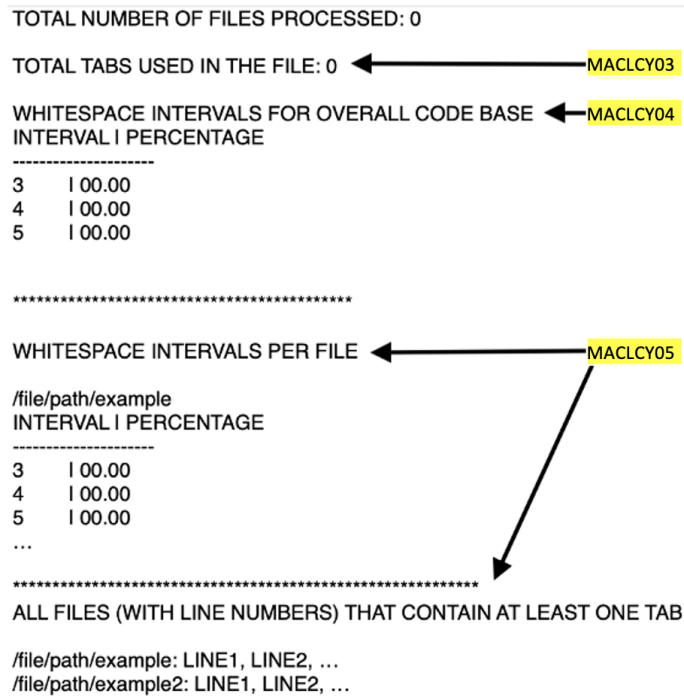


Figure 8.8.2 ConditionalCompiles.py sample output

```
Total Number of Unique Conditional Compiles Found: 0
Number of Conditional Compiles per file:
/Example/file/path1 -> 0
/Example/file/path2 -> 0
...
List of unique conditional compiles:
UNIQUE_CONDITIONAL_COMPILE1
UNIQUE_CONDITIONAL_COMPILE2
...
List of Conditional Compiles by file and their count:
/Example/file/path1
    UNIQUE_CONDITIONAL_COMPILE1 -> 0
    UNIQUE_CONDITIONAL_COMPILE2 -> 0
/Example/file/path2
    UNIQUE_CONDITIONAL_COMPILE1 -> 0
    UNIQUE_CONDITIONAL_COMPILE2 -> 0
...
```

The diagram shows two yellow boxes on the right side of the output. The box labeled 'MACLCD01' has an arrow pointing to the line 'Total Number of Unique Conditional Compiles Found: 0'. The box labeled 'MACLCD02' has an arrow pointing to the line 'List of Conditional Compiles by file and their count:'.

9 Maintainability Reusability

9.1 Overview

Recall, maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [IEEE 1990]. This tab analyzes the reusability of the source code, and whether its construction increases the overall risk.

To receive a satisfactory score for this tab, the code should have a high level of machine, OS, database, and compiler independence. The code should be portable to many different platforms, and platform specific types should not be mixed throughout the code. There should not be any unused variables, unused functions, or commented out, no-effect code. Lastly, there should be a low level of complicated implementation constructs, the maximum control nesting structure should be small, and there should be minimal duplicate code.

Many scripts are provided to automate and simplify parts of this analysis. SA tools also play a role in scoring this tab, but the direct visual inspection of source files is often needed to answer many of these questions. In some situations, when there is a wealth of data to analyze (i.e., script output results, all the source files, etc.), sampling is often employed to help narrow and generalize information for input into CREW. In general, there are 2 types of sampling used in this section. The first is sampling of SA output, as described in the relevant sections below. The second type of sampling is sampling script output, where the analyst will spot-check ~10% of the script output to generate the answer for CREW. The Maintainability Reusability tab is shown in Figure 9.1.1.

Figure 9.1.1 The Maintainability Reusability Tab:

Focus	Risk	W	ID	I	Measurement / Question	A	Finding	D	Note
INDEPENDENCE	3.1	0.250	MAREIN01	1	Does the code utilize in-line assembly language, e.g. the "asm" keyword?		No	0	
			MAREIN02	1	Does the code use other machine dependent constructs, e.g. "register" variables?		Few	0	
			MAREIN03	1	Does the code make assignments between integers, longs, and pointers?		Some	0	
			MAREIN04	1	Does the code mix signed and unsigned integers within expressions?		Many	0	
			MAREIN05	1	Does the code make use of bitfields?		Yes	0	
			MAREIN06	1	Does the code use subsystems to hide implementation details for OS, database, BSP, or device access?		No	0	
			MAREIN07	1	Are there time delays within the code that are built with simple for or while loop counters?		Some	0	
			MAREIN08	1	Does the code protect global variables within interrupt routines or within multiple threads?		No	0	
			MAREIN09	1	Does the code only utilize built-in platform-specific compiler flags, e.g. #ifdef _WIN32 rather than custom ones?		Yes	0	
			MAREIN10	1	Are system calls embedded directly within the code?		No	0	
			MAREIN11	1	Does the code utilize constants instead of sizeof() to encode storage sizes within expressions?		Some	0	
			MAREIN12	0	Is the code meant to build on a specific family of processors only?		No	0	
			MAREIN13	1	Does the code contain constructs that implicitly assume a particular data model (i.e. data type size) such as ILP32?	C	Yes	0	
			MAREIN14	1	Does the code contain constructs that are specific to big or little endian byte ordering?	C	Some	0	
			MAREIN15	0	Does the code utilize any 3rd-party or auto-generated source code?		Yes	0	
			MAREIN16	1	Is 3rd-party or auto-generated code intermixed along with custom code within source files?	C	Some	0	
			MAREIN17	1	Is 3rd-party or auto-generated code readily identifiable as such (whether in intermixed files or isolated files / sub-folders)?	C	No	0	
UTILIZATION	3.4	0.250	MAREUT01	1	Estimated total number of commented-out lines of code		• 1035	0	
			MAREUT02	1	Are commented-out lines interspersed freely among uncommented-out lines of code?		Some	0	
			MAREUT03	1	Estimated total number of unreachable statements within the code		20	0	
			MAREUT04	1	Estimated total number of unused functions or methods within the code		10	0	
			MAREUT05	1	Estimated total number of unused global variables within the code		10	0	
			MAREUT06	1	Estimated total number of unused local variables within the code		10	0	
			MAREUT07	1	Estimated total number of no-effect or redundant statements within the code		10	0	
			MAREUT08	1	Estimated percent of dead code within the code base		• 0.17	0	
COMPLICATION	4.0	0.250	MARECM01	1	Maximum control structure nesting level within the code		• 10	0	
			MARECM02	0	Total number of functions or methods within the code base		• 8157	0	
			MARECM03	1	Percentage of functions or methods within the code with Hersteller (HIS) VOCCF (LanguageScope) exceeding 4.0		10.0%	0	
			MARECM04	1	Number of goto occurrences within the code, i.e. Hersteller GOTO metric		10	0	
			MARECM05	1	Number of functions within the code with Hersteller AP_CYCLE (Recursion) of 1 or more		5	0	
			MARECM06	1	Percentage of functions or methods within the code with Hersteller CALLING (FanIn) exceeding 5		10.0%	0	
			MARECM07	1	Percentage of functions or methods within the code with Hersteller CALLS (FanOut) exceeding 7		10.0%	0	
			MARECM08	1	Percentage of functions or methods within the code with Hersteller PARAM (NumParams) exceeding 5		10.0%	0	
			MARECM09	1	Percentage of functions or methods within the code with Hersteller LEVEL (call nesting) exceeding 4		10.0%	0	
			MARECM10	1	Average Koopman Spaghetti Factor (KSF) of all implementation files		5.0	0	
			MARECM11	1	Percentage of implementation files with KSF exceeding 15		5.0%	0	
DUPLICATION	4.6	0.250	MAREDU01	1	What is the level of duplicated code lines, i.e. how much copy / pasted code is there?		Some	0	
			MAREDU02	0	Estimated Number of source files that contain any duplicated code		1000	0	
			MAREDU03	1	Estimated percentage of source files that contain any duplicated code		• 53.13%	0	
			MAREDU04	0	Source lines of code ignoring blank and comment lines (PSLOC)		• 382397	0	
			MAREDU05	1	Total number of lines of duplicated code detected		• 36744	0	
			MAREDU06	1	Duplication ratio (as a percentage) = total number of detected duplicated lines to PSLOC		• 9.6%	0	
			MAREDU07	1	Total Number of Files that were completely identical		10	0	
			MAREDU08	0	Total Number of sets of files (matching groups of files) that are identical		100	0	
			MAREDU09	0	Total Number of Lines in files that are found to be identical		10000	0	
			MAREDU10	0	Total Number of Files that contain at least one set of duplicated lines within itself		100	0	
			MAREDU11	1	Estimated Percentage of Files that contain at least one set of duplicate lines within itself		• 5.31%	0	
RISK	3.77								

The following sections describe how to score the Maintainability Reusability tab of CREW:

9.2 Independence

While all the questions in the Independence focus can be answered manually using eyes-on-code or using other analytic scripts, static analysis tools can add additional effectiveness and efficiency to the process. Ideally, several static analyzers would be used, and their outputs would be aggregated to provide a more complete picture of potential risks in the code base. Additionally, a single question may have several static analyzer checkers that correspond to it within the same tool.

In the following line items, we utilize Parasoft, but other SA tools can be used based on existing organizational constraints to identify potential findings in relation to the line items below. Within the applicable line items below, there is a list of Parasoft checkers that were identified to have analyzed the code for risks and bugs associated with that line item. A test configuration containing those checkers was created and run across the code base, and a post-processing script was used to aggregate each checker's findings and map the results to a CREW line item. An analyst should select ~10% of the findings and adjudicate their validity (as SA tools are known to generate false positives). From there, the line items are filled in using the "scores" table associated with each line item.

A sample of the script output for the mapping of checkers to CREW line items can be seen in 9.6.1 and 9.6.2. The test configurations used can be provided upon request.

MAREIN01: Does the code utilize in-line assembly language, e.g., the "asm" keyword?

EXEMPLAR TOOL(S):

KeywordSearch.py and Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Use the output of the scripts KeywordSearch.py to answer the question (the “STANDARDIZED CODE MODULE KEYWORD” section). See 9.6.3 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives. Additionally, run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: AUTOSAR-A7_4_1-a, AUTOSAR-M7_4_2-a, AUTOSAR-M7_4_3-a, MISRAC2012-DIR_4_3-a, OWASP2019-API9-c]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives. Additionally, for the SA results, follow the process laid out at the top of this section. The findings in this case will help identify potential uses of assembly language within a code base.

SCORES:

No	Never uses assembly language
Few	Uses assembly language <2 times
Some	Uses assembly language <5 times
Many	Uses assembly language <10 times
Yes	Uses assembly language >10 times

MAREIN02: Does the code use other machine dependent constructs, e.g., "register" variables?

EXEMPLAR TOOL(S):

KeywordSearch.py and Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Use the output of the scripts KeywordSearch.py to answer the question (the “HARDWARE DEPENDENT KEYWORD” section). See 9.6.3 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives. Additionally, run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: AUTOSAR-M0_1_4-a, CERT_C-CON02-a, HICPP-1_3_2-a]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives. Additionally, for the SA results, follow the process laid out at the top of this section. The findings in this case will help identify potential uses of machine dependent constructs within a code base.

SCORES:

No	Never uses machine dependent constructs
Few	Uses machine dependent constructs <10 times
Some	Uses machine dependent constructs <50 times
Many	Uses machine dependent constructs <100 times

Yes	Uses machine dependent constructs >100 times
-----	--

MAREIN03: Does the code make assignments between integers, longs, and pointers?

EXEMPLAR TOOL(S):

Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: AUTOSAR-M5_2_9-a, AUTOSAR-M5_2_8-a, CERT_C-ARR39-c, CERT_C-INT02-b, PORT-05, PORT-06, PORT-07, PORT-08, PORT-10, PORT-11, PORT-12, PORT-14, PORT-15, PORT-16, PORT-19, PORT-20, PORT-21, PORT-22, PORT-23, PORT-24]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Additionally, for the SA results, follow the process laid out at the top of this section to determine whether the findings are true or false positives.

SCORES:

No	No true positives
Few	<25% true positives
Some	<50% true positives
Many	<75% true positives
Yes	>75% true positives

MAREIN04: Does the code mix signed and unsigned integers within expressions?

EXEMPLAR TOOL(S):

Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: AUTOSAR-A4_7_1-a, AUTOSAR-M5_0_4-a, JSF-162_a, JSF-162_b, JSF-162_c, PORT-13]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Additionally, for the SA results, follow the process laid out at the top of this section to determine whether the findings are true or false positives.

SCORES:

No	No true positives
Few	<25% true positives
Some	<50% true positives
Many	<75% true positives
Yes	>75% true positives

MAREIN05: Does the code make use of bitfields?

EXEMPLAR TOOL(S):

Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: AUTOSAR-A9_6_2-a, AUTOSAR-M9_6_4-a, CERT_C-INT12-a, CWE-362-c, HICPP-9_2_1-a, MISRA2004-3_5, MISRA2004-6_5_a, MISRA2008-9_6_2, MISRA2008-9_6_3]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives. Additionally, for the SA results, follow the process laid out at the top of this section. The findings in this case will help identify potential uses of bit fields within a code base. These rules will highlight static analysis issues related to bit fields, which would imply the presence of them.

SCORES:

No	No true positives
Few	<25 instances of bit fields found
Some	<75 instances of bit fields found
Many	<100 instances of bit fields found
Yes	>100 instances of bit fields found

MAREIN06: Does the code use subsystems to hide implementation details for OS, database, BSP, or device access?

EXEMPLAR TOOL(S):

KeywordSearch.py.

PROCESS:

Run the script KeywordSearch.py and utilize the output to answer the question (the “BSP OS DATABASE KEYWORD” section). The script looks for those keywords listed under KeywordsList.py and can produce false positives. See 9.6.3 for utilization of script output. The script can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives.

SCORES:

No	>3 true positives of the same keyword in a non-abstracted method
Some	<3 true positives of the same keyword in a non-abstracted method
Yes	All instances of the keywords are in an abstracted method

MAREIN07: Are there time delays within the code that are built with simple for or while loop counters?

EXEMPLAR TOOL(S):

KeywordSearch.py.

PROCESS:

Run the script KeywordSearch.py and utilize the output to answer the question (the “DELAY KEYWORD” section). The script looks for those keywords listed under KeywordsList.py and can produce false positives. See 9.6.3 for utilization of script output. The script just looks for keywords and can produce false positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives.

SCORES:

No	No evidence of using non-standard delay methods
Some	<3 instances of non-standard delay methods
Yes	>3 instances of non-standard delay methods

MAREIN08: Does the code protect global variables within interrupt routines or within multiple threads?

EXEMPLAR TOOL(S):

KSF_Understand.py and Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Run the script KSF_Understand.py and utilize the output to answer the question. See 9.6.4 for utilization of script output. The script produces a list of used global variables. Additionally, run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: BD-TRS-DIFCS, CWE-362-e]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

The global variables identified will have to be reviewed individually to validate they are protected within interrupt routines or threads. Additionally, for the SA results, follow the process laid out at the top of this section to determine whether the findings are true or false positives. These findings will help identify unprotected global variables.

SCORES:

No	Global variables are used but are not protected.
Yes	Global variables are used and protected (or not used)

MAREIN09: Does the code only utilize built-in platform-specific compiler flags, e.g., #ifdef _WIN32 rather than custom ones?

EXEMPLAR TOOL(S):

KeywordSearch.py and Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Use the output of the scripts KeywordSearch.py to answer the question (the “OS MACRO KEYWORD” section). See 9.6.3 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives. Additionally, run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: CERT_CPP-DCL51-a]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Using the output, determine if all uses of the macro keywords are built-in, or custom macros. The list with differentiation can be found in KeywordLists.py. Additionally, for the SA results, follow the process laid out at the top of this section to determine whether the findings are true or false positives. The findings in this case will help identify custom vs. non-custom OS related macros.

SCORES:

No	The code uses at least some custom macros
Yes	The code uses built-in macros only

MAREIN10: Are system calls embedded directly within the code?

EXEMPLAR TOOL(S):

KeywordSearch.py and Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

Use the output of the scripts KeywordSearch.py to answer the question (the “UNSAFE LIBRARY OR OS CALL KEYWORD” section). See 9.6.3 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives. Additionally, run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: HICPP-17_2_1-a]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Using the output, identify the total number of lines with a system call being made. In general, the more instances that are found, the less likely that the class has been abstracted away. Additionally, for the SA results, follow the process laid out at the top of this section to determine whether the findings are true or false positives. The findings in this case will help identify system calls within code.

SCORES:

No	<25 system calls
Few	<100 system calls
Some	<400 system calls
Many	<600 system calls
Yes	>600 system calls

MAREIN11: Does the code utilize constants instead of sizeof() to encode storage sizes within expressions?

EXEMPLAR TOOL(S):

KeywordSearch.py.

PROCESS:

Run the script KeywordSearch.py and utilize the output to answer the question (the “SIZE KEYWORD” section). See 9.6.3 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives.

EVALUATION:

Using the output, identify the general distribution of instances of using sizeof() vs. using some other keywords that typically are indicative of variable length. The more uses of sizeof(), as opposed to some of the other variables, the more likely they don’t use constants.

SCORES:

No	The vast majority of instances found are using sizeof()
Some	The distribution seems to be split between sizeof() and other variants
Yes	The vast majority of the instances do not use sizeof()

MAREIN12: Is the code meant to build on a specific family of processors only?

EXEMPLAR TOOL(S):

N/A

PROCESS:

Based on conversations with the customer, or analyzing the code or the build process, determine whether the code is meant to be portable, or if it is meant to be built on a single build processor at a time.

EVALUATION:

Determine if the code is intended for a single processor family or is meant to be portable.

SCORES:

No	Code is not limited to a singular processor family
Yes	Code is limited to a singular processor family

MAREIN13: Does the code contain constructs that implicitly assume a particular data model (e.g. data type size) such as ILP32?

EXEMPLAR TOOL(S):

Static Analysis Tools (i.e., Understand)

PROCESS:

This question is conditional on MAREIN12. Using a static analysis tool, search for the user defined types used within the code base.

EVALUATION:

Going through those types, manually review at least 50% of how those types are defined, and whether the exact sizes are being specified for any user defined types. Additionally evaluate the consistency of those data models being followed.

SCORES:

No	No evidence of using a particular data model
Few	Most definitions do not follow a data model
Some	A data model is followed in only a portion of the definitions
Many	Most definitions follow a data model
Yes	Consistent use of a particular data model in definitions

MAREIN14: Does the code contain constructs that are specific to big or little endian byte ordering?

EXEMPLAR TOOL(S):

KeywordSearch.py and Static Analysis Tools (i.e., Parasoft/MapParasoftViolationsToCrew.py)

PROCESS:

This question is conditional on MAREIN12. Use the output of the scripts KeywordSearch.py to answer the question (the “HARDWARE DEPENDENT KEYWORD” section). See 9.6.3 for utilization of script output. The script looks for those keywords listed under KeywordsList.py and can produce false positives. Additionally, run a static analysis tool and perform the mapping (i.e., Parasoft/MapParasoftViolationsToCrew.py [Checkers: CERT_C-POS39-a]). Visual inspection is needed to verify if the results are true positives.

EVALUATION:

Using the output, one keyword at a time, follow the sampling process described in the introduction and determine whether these are true or false positives. The evaluation of the various endian related keywords will help determine whether endianness is in use. Additionally, for

the SA results, follow the process laid out at the top of this section to determine whether the findings are true or false positives. The findings in this case will help identify potential uses of endianness.

SCORES:

No	It does not appear to assume endianness
Some	There are endian related functions that are seldom used
Yes	Endianness is directly accounted for within the code

MAREIN15: Does the code utilize any 3rd-party or auto-generated source code?

EXEMPLAR TOOL(S):

Build Tools (native to code's build environment).

PROCESS:

This is a 2-step process. First is to look at any build scripts to see if any file generation is happening. Second, it is possible that file generation happens in a non-obvious manner. Analyze the source directory by looking for any files with timestamps that are marked after the build was created.

EVALUATION:

Look for new or updated files in the source directory. In most cases, at least one build directory is created. By looking for source files (.c, .cpp, .h, etc.), you can see if any files were created in this directory. Additionally, looking at different directory names, or potentially looking in any provided documentation should help determine if 3rd party software is used.

SCORES:

No	The code does not utilize auto-generated code or 3 rd party software
Yes	The code utilizes auto-generated code or 3 rd party software

MAREIN16: Is 3rd-party or auto-generated code intermixed along with custom code within source files?

EXEMPLAR TOOL(S):

Build Tools (native to code's build environment).

PROCESS:

This question is conditional on MAREIN15 and whether any files were found to be auto generated or be 3rd party.

EVALUATION:

Visually review ~10% of the files that are autogenerated or 3rd party to see if they are separated or intermixed within source files.

SCORES:

No	Autogenerated or 3 rd party files are separate
Few	Autogenerated or 3 rd party files are mostly separated with a few intermixed
Some	Autogenerated or 3 rd party files are split between separate and mixed
Many	Autogenerated or 3 rd party files are mostly intermixed with a few separate
Yes	Autogenerated or 3 rd party files are intermixed

MAREIN17: Is 3rd-party or auto-generated code readily identifiable as such (whether in intermixed files or isolated files / sub-folders)?

EXEMPLAR TOOL(S):

Build Tools (native to code's build environment).

PROCESS:

This question is conditional on MAREIN15 and whether any files were found to be auto generated or be 3rd party.

EVALUATION:

Visually review ~10% of the files that are autogenerated or 3rd party to see if they are readily identifiable.

SCORES:

No	The autogenerated or 3 rd party software is not readily identifiable
Some	Some, but not all the files are readily identifiable
Yes	The autogenerated or 3 rd party software is readily identifiable

9.3 Utilization

MAREUT01: Estimated total number of commented-out lines of code

From System tab, Metrics Focus, Item #22.

MAREUT02: Are commented-out lines interspersed freely among uncommented-out lines of code?

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide information on commented out code (i.e., Understand CodeCheck RECOMMEND_00 or CPP_C000).

EVALUATION:

Using the output, visually review ~20% of the identified commented out lines of code to determine if the commented-out code is segregated from non-commented out sections.

SCORES:

No	All commented out code appears to be separated from active code
Few	Most of the commented-out code appears to be separate from active code
Some	There appears to be even split between being separate and interspersed
Many	Most of the commented-out code appears to be mixed with active code
Yes	All commented out code appears to be interspersed with active code

MAREUT03: Estimated total number of unreachable statements within the code

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide information on unreachable statements (i.e., Understand CodeCheck RECOMMEND_12 or CPP_C034).

EVALUATION:

Using the output, visually review ~10% of the identified unreachable code to verify if the results are true positives.

SCORES:

Take the direct output as the score.

MAREUT04: Estimated total number of unused functions or methods within the code

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide information on unused functions or methods (i.e., Understand CodeCheck RECOMMEND_13 or CPP_F003).

EVALUATION:

Using the output, visually review ~10% of the identified unused functions or methods to verify if the results are true positives.

SCORES:

Take the direct output as the score.

MAREUT05: Estimated total number of unused global variables within the code

EXEMPLAR TOOL(S):

KSF_Understand.py

PROCESS:

Run the script KSF_Understand.py and utilize the output to answer the question.

EVALUATION:

See 9.6.5 for utilization of script output.

SCORES:

Take the direct output as the score.

MAREUT06: Estimated total number of unused local variables within the code

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide information on unused local variables (i.e., Understand CodeCheck RECOMMEND_14 or CPP_V007).

EVALUATION:

Using the output, visually review ~10% of the identified unused local variables to verify if the results are true positives.

SCORES:

Take the direct output as the score.

MAREUT07: Estimated total number of no-effect or redundant statements within the code

EXEMPLAR TOOL(S):

RedundantCode.py or Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Run the script RedundantCode.py and utilize the output to answer the question. Additionally, static analysis tools can provide information on no-effect (unreachable) code (i.e., Understand CodeCheck RECOMMEND_12 or CPP_C034).

EVALUATION:

See 9.6.6 for utilization of script output. Additionally using the static analysis tool output, visually review ~10% of the identified no-effect (unreachable) code to verify if the results are true positives.

SCORES:

Take the sum of the outputs from the static analysis tools and scripts.

MAREUT08: Estimated percent of dead code within the code base

Calculated by using the following equation (done automatically in spreadsheet):

$$\frac{TOTAL_COMMENTED_OUT_LOC + TOTAL_REDUNDANT_LOC + TOTAL_NO_EFFECT_REDUNDANT_LOC}{PSLOC} \times 100$$

9.4 Complication

MARECM01: Maximum control structure nesting level within the code

From System tab, Metrics Focus, Item #19.

MARECM02: Total number of functions or methods within the code base

From System tab, Metrics Focus, Item #16.

MARECM03: Percentage of functions or methods within the code with Hersteller (HIS) VOF (LanguageScope) exceeding 4.0

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_11).

EVALUATION:

Using the output from the static analysis tool, calculate the following:

$$\frac{NUMBER_OF_FUNCTIONS_WITH_VOF_EXCEEDING_4}{TOTAL_NUMBER_OF_FUNCTIONS_OR_METHODS} \times 100\%$$

SCORES:

Take the direct output as the score.

MARECM04: Number of goto occurrences within the code, e.g. Hersteller GOTO metric

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_03).

EVALUATION:

Using the output, visually review ~10% to verify if the results are true positives.

SCORES:

Take the direct output as the score.

MARECM05: Number of functions within the code with Hersteller AP_CG_CYCLE (Recursion) of 1 or more

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_12).

EVALUATION:

Using the output, visually review ~10% to verify if the results are true positives.

SCORES:

Take the direct output as the score.

MARECM06: Percentage of functions or methods within the code with Hersteller CALLING (FanIn) exceeding 5

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_05).

EVALUATION:

Using the output from the static analysis tool, calculate the following:

$$\frac{NUMBER_OF_FUNCTIONS_WITH_FANIN_EXCEEDING_4}{TOTAL_NUMBER_OF_FUNCTIONS_OR_METHODS} \times 100\%$$

SCORES:

Take the direct output as the score.

MARECM07: Percentage of functions or methods within the code with Hersteller CALLS (FanOut) exceeding 7

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_06).

EVALUATION:

Using the output from the static analysis tool, calculate the following:

$$\frac{NUMBER_OF_FUNCTIONS_WITH_FANOUT_EXCEEDING_7}{TOTAL_NUMBER_OF_FUNCTIONS_OR_METHODS} \times 100\%$$

SCORES:

Take the direct output as the score.

MARECM08: Percentage of functions or methods within the code with Hersteller PARAM (NumParams) exceeding 5

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_07).

EVALUATION:

Using the output from the static analysis tool, calculate the following:

$$\frac{NUMBER_OF_FUNCTIONS_WITH_NUMPARAMS_EXCEEDING_5}{TOTAL_NUMBER_OF_FUNCTIONS_OR_METHODS} \times 100\%$$

SCORES:

Take the direct output as the score.

MARECM09: Percentage of functions or methods within the code with Hersteller LEVEL (call nesting) exceeding 4

EXEMPLAR TOOL(S):

Static Analysis Tool (e.g., SciTools Understand).

PROCESS:

Static analysis tools can provide this information (i.e., Understand CodeCheck HIS_09).

EVALUATION:

Using the output from the static analysis tool, calculate the following:

$$\frac{NUMBER_OF_FUNCTIONS_WITH_CALL_NESTING_EXCEEDING_4}{TOTAL_NUMBER_OF_FUNCTIONS_OR_METHODS} \times 100\%$$

SCORES:

Take the direct output as the score.

MARECM10: Average Koopman Spaghetti Factor (KSF) of all implementation files

EXEMPLAR TOOL(S):
KSF_Understand.py.

PROCESS:
Run the script KSF_Understand.py and utilize the output to answer the question.

EVALUATION:
See Figure 9.6.7 for utilization of script output.

SCORES:
Take the direct output as the score.

MARECM11: Percentage of implementation files with KSF exceeding 15

EXEMPLAR TOOL(S):
KSF_Understand.py.

PROCESS:
Run the script KSF_Understand.py and utilize the output to answer the question.

EVALUATION:
See Figure 9.6.8 for utilization of script output. There are 2 ways to calculate this percentage, either subtracting from 100% the sum of those percentages of files with a KSF score less than 10 or sum all the percentages for files with a KSF score of more than 15.

SCORES:
Take the direct output as the score.

9.5 Duplication

MAREDU01: What is the level of duplicated code lines, e.g. how much copy / pasted code is there?

EXEMPLAR TOOL(S):
N/A.

PROCESS:
Interpretation of results from answering MAREDU02- MAREDU05.

EVALUATION:
Based on the answers, and the level of duplication, qualify the results from the rest of the duplication focus.

SCORES:

None	0% code duplication found in files and lines
Some	<0.5% of lines are duplicated, and/or <1% of files are duplicated
Many	>0.5% of lines and/or >1% of files are duplicated

MAREDU02: Estimated Number of source files that contain any duplicated code

EXEMPLAR TOOL(S):
Duplicate_code.upl and DuplicateCodePostProcessing.py

PROCESS:

Run the duplicate_code.upl, which is a plugin for SciTools Understand, then run the output through the script DuplicateCodePostProcessing.py.

EVALUATION:

See Figure 9.6.9 for utilization of script output.

SCORES:

Take the direct output as the score

MAREDU03: Estimated percentage of source files that contain any duplicated code

Automatically calculated from other numbers in the spreadsheet using the following calculation:

$$\frac{NUMBER_OF_FILES_THAT_CONTAIN_DUPLICATION}{TOTAL_NUMBER_OF_FILES} \times 100\%$$

MAREDU04: Source lines of code ignoring blank and comment lines (PSLOC)

From System tab, Metrics Focus, Item #12.

MAREDU05: Total number of lines of duplicated code detected

Duplicate_code.upl and DuplicateCodePostProcessing.py

PROCESS:

Run the duplicate_code.upl, which is a plugin for SciTools Understand, then run the output through the script DuplicateCodePostProcessing.py.

EVALUATION:

See Figure 9.6.9 for utilization of script output.

SCORES:

Take the direct output as the score.

MAREDU06: Duplication ratio (as a percentage) = total number of detected duplicated lines to PSLOC

Automatically calculated from other numbers in the spreadsheet using the following calculation:

$$\frac{NUMBER_OF_LINES_DUPLICATED}{PSLOC} \times 100\%$$

MAREDU07: Total Number of Files that were completely identical

Duplicate_code.upl and DuplicateCodePostProcessing.py

PROCESS:

Run the duplicate_code.upl, which is a plugin for SciTools Understand, then run the output through the script DuplicateCodePostProcessing.py.

EVALUATION:

See Figure 9.6.9 for utilization of script output.

SCORES:

Take the direct output as the score.

MAREDU08: Total Number of sets of files (matching groups of files) that are identical

Duplicate_code.upl and DuplicateCodePostProcessing.py

PROCESS:

This is the number of sets of identically duplicated files. For example, say that File A and File B are identical, that is one set, if File C, D, and E, are identical, that would be another set. So there would be 5 duplicated files across 2 sets. Run the duplicate_code.upl, which is a plugin for SciTools Understand, then run the output through the script DuplicateCodePostProcessing.py.

EVALUATION:

See Figure 9.6.9 for utilization of script output.

SCORES:

Take the direct output as the score.

MAREDU09: Total Number of Lines in files that are found to be identical.

Duplicate_code.upl and DuplicateCodePostProcessing.py

PROCESS:

Run the duplicate_code.upl, which is a plugin for SciTools Understand, then run the output through the script DuplicateCodePostProcessing.py.

EVALUATION:

See Figure 9.6.9 for utilization of script output.

SCORES:

Take the direct output as the score.

MAREDU10: Total Number of Files that contain at least one set of duplicated lines within itself.

Duplicate_code.upl and DuplicateCodePostProcessing.py

PROCESS:

Run the duplicate_code.upl, which is a plugin for SciTools Understand, then run the output through the script DuplicateCodePostProcessing.py.

EVALUATION:

See Figure 9.6.9 for utilization of script output.

SCORES:

Take the direct output as the score.

MAREDU11: Estimated Percentage of Files that contain at least one set of duplicate lines within itself

Automatically calculated from other numbers in the spreadsheet using the following calculation:

$$\frac{NUMBER_OF_FILES_THAT_CONTAIN_DUPLICATION_WITHIN_ITSELF}{TOTAL_NUMBER_OF_FILES} \times 100\%$$

9.6 Script Outputs/Examples

Figure 9.6.1 MapParasoftViolationsToCrew.py condensed output example

```
1 *****
2
3  ✓ CREW_RULE_NUMBER_1 : 0 ← Total Parasoft Findings for CREW rule (sum of all checkers)
4      PARASOFT_CHECKER_ID_1 : 0
5      PARASOFT_CHECKER_ID_2 : 0
6      PARASOFT_CHECKER_ID_3 : 0 ← Findings of this checker
7      ...
8      PARASOFT_CHECKER_ID_N : 0
9  ✓ CREW_RULE_NUMBER_2 : 0
10     PARASOFT_CHECKER_ID_1 : 0
11     PARASOFT_CHECKER_ID_2 : 0
12     PARASOFT_CHECKER_ID_3 : 0
13     ...
14     PARASOFT_CHECKER_ID_N : 0
15     ...
16  ✓ CREW_RULE_NUMBER_N : 0
17     PARASOFT_CHECKER_ID_1 : 0
18     PARASOFT_CHECKER_ID_2 : 0
19     PARASOFT_CHECKER_ID_3 : 0
20     ...
21     PARASOFT_CHECKER_ID_N : 0
22
23 *****
24
```

Figure 9.6.2 MapParasoftViolationsToCrew.py full output with all findings example

```

1 *****
2
3 CREW_RULE_NUMBER_1 : 0
4   PARASOFT_CHECKER_ID_1 : 0
5     FILE_NAME_1:LINE_NUMBER -> VIOLATION MESSAGE ← Individual finding/location for adjudication
6     FILE_NAME_2:LINE_NUMBER -> VIOLATION MESSAGE
7     FILE_NAME_3:LINE_NUMBER -> VIOLATION MESSAGE
8     ...
9     FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
10  PARASOFT_CHECKER_ID_2 : 0
11    FILE_NAME_1:LINE_NUMBER -> VIOLATION MESSAGE
12    FILE_NAME_2:LINE_NUMBER -> VIOLATION MESSAGE
13    FILE_NAME_3:LINE_NUMBER -> VIOLATION MESSAGE
14    ...
15    FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
16  PARASOFT_CHECKER_ID_3 : 0
17    ...
18    FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
19  ...
20  PARASOFT_CHECKER_ID_N : 0
21    ...
22    FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
23  ...
24  CREW_RULE_NUMBER_N : 0
25    PARASOFT_CHECKER_ID_1 : 0
26      ...
27      FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
28    PARASOFT_CHECKER_ID_2 : 0
29      ...
30      FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
31    PARASOFT_CHECKER_ID_3 : 0
32      ...
33      FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
34    ...
35    PARASOFT_CHECKER_ID_N : 0
36      ...
37      FILE_NAME_N:LINE_NUMBER -> VIOLATION MESSAGE
38  *****
39
40

```

Figure 9.6.3 KeywordSearch.py sample output

```

1  *****
2
3
4  TOTAL NUMBER OF LINES PROCESSED: 0
5  TOTAL NUMBER OF LINES WITH A <CATEGORY> KEYWORD: 0
6  LINES WITH A <CATEGORY> KEYWORD (%): 0
7
8
9  *****
10 ALL <CATEGORY> KEYWORDS AND THE NUMBER OF TIMES EACH APPEAR PER FILE AND WHERE IN EACH FILE
11
12  ✓ KEYWORD #1 [NUMBER_OF_OCCURANCES]
13      FILE_1 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
14      FILE_2 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
15      FILE_3 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
16      ...
17      FILE_N [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
18  ✓ KEYWORD #2 [NUMBER_OF_OCCURANCES]
19      FILE_1 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
20      FILE_2 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
21      FILE_3 [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
22      ...
23      FILE_N [NUMBER_OF_OCCURANCES]: LINE_NUMBER_1 LINE_NUMBER_2 LINE_NUMBER_3
24
25

```

Figure 9.6.4 KSF_Understand.py used global variable sample output

```

1  USED GLOBAL VARIABLES:
2
3  TOTAL NUMBER OF USED GLOBAL VARIABLES: 0
4  VARIABLE_NAME_1
5  VARIABLE_NAME_2
6  VARIABLE_NAME_3
7  ...
8  VARIABLE_NAME_N
9

```


Evaluate each instance and ensure its protection based on use
MAREIN08

Figure 9.6.5 KSF_Understand.py unused global variable sample output

```

1  UNUSED GLOBAL DATA:
2
3  TOTAL NUMBER OF UNUSED VARIABLES: 0 ← MAREUT05
4  VARIBALE_NAME_1
5  VARIABLE_NAME_2
6  VARIABLE_NAME_3
7  ...
8  VARIABLE_NAME_N
9

```

Figure 9.6.6 RedundantCode.py sample output

```

1  *****
2  TOTAL REDUNDANT LINES: 0 ← MAREUT07
3  *****
4

```

Figure 9.6.7 KSF_Understand.py KSF file distribution sample output

```

1  THE AVERAGE KSF FOR ALL FILES PROCESSED ( NUMBER_OF_SOURCE_FILES ) IS: 0 ← MARECM10
2  -----
3  File                                     Complexity  Globals  Lines  KSF
4  -----
5  FILE_WITH_LARGEST_KSF_SCORE              0           0       0      0
6  FILE_WITH_SECOND_LARGEST_KSF_SCORE        0           0       0      0
7  FILE_WITH_THIRD_LARGEST_KSF_SCORE         0           0       0      0
8  FILE_WITH_FOURTH_LARGEST_KSF_SCORE        0           0       0      0
9  ...
10 FILE_WITH_NTH_LARGEST_KSF_SCORE           0           0       0      0
11

```

Figure 9.6.8 KSF_Understand.py KSF histogram data sample output

```

1  HISTOGRAM DATA:
2
3  FILES WITH KSF LESS THAN 10:  0 or 0%
4  FILES WITH KSF LESS THAN 15:  0 or 0%
5  FILES WITH KSF LESS THAN 20:  0 or 0%
6  FILES WITH KSF LESS THAN 30:  0 or 0%
7  FILES WITH KSF LESS THAN 40:  0 or 0%
8  FILES WITH KSF LESS THAN 50:  0 or 0%
9  FILES WITH KSF GREATER THAN 50: 0 or 0%
10

```

MARECM11: 100% - Sum of these percentages

Figure 9.6.8 DuplicateCodePostProcessing.py sample output

```

1  *****
2
3  TOTAL DUPLICATE LINES: 0 ← MAREDU05
4  TOTAL NUMBER OF FILES THAT CONTAIN DUPLICATE LINES: 0 ← MAREDU02
5
6  TOTAL NUMBER OF FILES THAT WERE FOUND TO BE IDENTICAL: 0 ← MAREDU07
7  TOTAL NUMBER OF SETS OF FILES (MATCHING GROUPS) THAT ARE IDENTICAL: 0 ← MAREDU08
8  TOTAL NUMBER OF LINES WITH FILES THAT ARE IDENTICAL: 0 ← MAREDU09
9  PERCENTAGE OF DUPLICATE LINES THAT ARE IN IDENTICAL FILES (%): 0.0
10
11 TOTAL NUMBER OF FILES THAT HAVE DUPLICATE LINES WITHIN ITSELF: 0 ← MAREDU10
12
13 *****
14

```

10 Fault Tolerance

10.1 Overview

This tab captures the risk factors associated with the fault tolerance capacity of the system. Fault tolerance reflects the ability of software to detect and recover from faults, whether they are external (e.g., failed h/w or improper inputs) or internal (e.g., incorrect code). Fault tolerant code safeguards proper operation and takes appropriate action based on the nature of the failure and its essential quality attributes (QA). For example, a safety-critical (QA = Safety) application must transition to a safe state, while a software system that must remain operational (QA = Availability) may require redundant elements. In general, fault tolerant code verifies all inputs and outputs both at the system's perimeter and internally, uses consistent error handling and logging, and continuously monitors itself and external components for proper operation.

To receive a satisfactory score for the Fault Tolerance tab, the code base should exhibit a high level of return and parameter value checking, and consistently check for and return proper input and output values. The code should also contain a systematic error handling mechanism that manages all errors and exceptions. There should also be an error/fault logging mechanism that is consistently used. Furthermore, a mechanism for capturing run-time statistics and timing information should be present, and the code should monitor and record average and worst-case run-time characteristics. Lastly, the code should effectively use system health monitoring tools, techniques, and procedures (e.g. watchdog timers) and attempt to restart the system if necessary. The Fault Tolerance tab is shown in Figure 10.1.1.

Figure 10.1.1 The Fault Tolerance Tab

Focus	Risk	W	ID	I	Measurement / Question	A	Finding	D
VALUE CHECKING	3.1	0.300	FLTOVC01	1	Are function return values checked consistently?		Few	0
			FLTOVC02	1	Estimated number of calls without the return value checked		100	0
			FLTOVC03	1	Are there instances of unchecked return values with subsequent usage?		No	0
			FLTOVC04	1	Are there instances where all possible return values from a function are not accounted for?		Few	0
			FLTOVC05	1	Are there instances of functions not returning a value or throwing an exception on an error?		Some	0
			FLTOVC06	1	Are pointers confirmed to be correctly initialized and non-null before use?		Many	0
			FLTOVC07	1	Are mem alloc & dealloc mismatched (delete vs delete[], new + free, realloc failure)?		Some	0
			FLTOVC08	1	Are memory allocations correctly handled in error blocks (try/catch in new, NULL in malloc)?		No	0
			FLTOVC09	1	Are parameter values within functions confirmed to be within expected bounds before use?		Yes	0
			FLTOVC10	1	Are user inputs confirmed to be of the correct type?		No	0
			FLTOVC11	1	Are user inputs confirmed to be within an expected range?		Few	0
ERROR HANDLING	2.8	0.300	FLTOEH01	1	Is there evidence of a systematic error handling mechanism?		Some	0
			FLTOEH02	1	Do functions / methods consistently return an error or throw an exception on failure?		Yes	0
			FLTOEH04	1	Is there evidence of a consistent error-handling pattern?		No	0
ERROR LOGGING	3.9	0.200	FLTOEL01	1	Are errors consistently logged?		No	0
			FLTOEL02	1	Are exceptional conditions consistently logged?		Few	0
			FLTOEL03	1	Are security events consistently logged?		Some	0
			FLTOEL04	1	Are logs sufficiently detailed to identify the issue?		No	0
			FLTOEL05	1	Are stored logs maintained to not exceed a maximum allowable size?		Some	0
INSTRUMENTATION	3.2	0.100	FLTOIN01	1	Is there evidence of instrumentation at all?		No	0
			FLTOIN02	1	Are run-time communication channels utilization recorded?		Some	0
			FLTOIN03	1	Are worst-case and average run-time memory utilization recorded?		Yes	0
			FLTOIN04	1	Are worst-case and average run-time timing latencies recorded?		No	0
			FLTOIN05	1	Are boot-up (turn on) counts recorded?		No	0
			FLTOIN06	1	Are system (total) run-time hours recorded?		Yes	0
			FLTOIN07	1	Are worst-case and average run-time processor utilization recorded?		Some	0
HEALTH MONITORING	5.0	0.100	FLTOHM01	1	Does the system utilize a watchdog timer?		No	0
			FLTOHM02	1	Does the code verify the health of all tasks and threads before strobing the watchdog?		No	0
			FLTOHM03	1	Are fault and watchdog reboot counts recorded?		No	0
			FLTOHM04	1	Are system health and instrumentation data accessible to system administrators?		No	0
RISK	3.38							

The following sections describe how to complete the CREW Fault Tolerance tab:

10.2 Value Checking

FLTOVC01: Are function return values checked consistently?

EXEMPLAR TOOL(S):

clang-tidy, CppCheck, SonarQube, etc.

PROCESS:

Filter rules to identify whether there are possible return values that are not checked. Verify in source code that the tool has not identified a false positive.

EVALUATION:

Number of occurrences where return values are not checked (v) over source lines of code (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$x \geq 0.4\%$
Few	Medium High	$0.3\% \leq x < 0.4\%$
Some	Medium	$0.2\% \leq x < 0.3\%$
Most	Medium Low	$0.1\% \leq x < 0.2\%$
Yes	Low	$0\% \leq x < 0.1\%$

FLTOVC02: Estimated number of calls without the return value checked

EXEMPLAR TOOL(S):

clang-tidy, CppCheck, SonarQube, etc.

PROCESS:

Evaluate the question above and the question immediately following this one regarding subsequent usage prior to answering this question. Collect the raw numbers in both circumstances, if possible (not all tools check for subsequent usage).

EVALUATION:

Enter in the number. The number is not always straightforward, in that, analysts will need to account for subsequent usage that does not overlap with unused return values. The result is often a rounded number for simplicity, along with a note.

SCORES:

Non-negative integer

FLTOVC03: Are there instances of unchecked return values with subsequent usage?

EXEMPLAR TOOL(S):

clang-tidy, CppCheck, SonarQube, etc. and manual sampling.

PROCESS:

Many static-analysis tools neglect to provide subsequent usage unless it's a possible null dereference. The process requires checking for those possible null-dereferences and then sampling the number of other ignored return values to determine pervasiveness of the problem.

Manual sampling is required for non-null variables (primitives, pass-by-value, pass-by-reference). Manual sampling takes place throughout the duration of the analysis project by capturing violations while searching for and validating other potential violations.

EVALUATION:

Calculate the number of violations (v) over the number of files checked (n), where $v / n = x$

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC04: Are there instances where all possible return values from a function are not accounted for?

EXEMPLAR TOOL(S):

Mostly manual. Clang-tidy and other tools help provide potential sections of code to begin checking, such as uncaught exceptions, but they do not typically explicitly state that error pathways are unaccounted for outside of try/catch paradigms.

PROCESS:

Search across the code base for error patterns in methods, e.g., try/catch, error numbers, error logging. Find references to methods that have error pathways.

EVALUATION:

If the client code fails to check error pathways, i.e., client code does not manage error outcomes, such as thrown or rethrown exceptions. $x = v / \text{files checked}$

SCORES:

CREW	Risk	Description
No	High	$x \geq 4\%$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 0.1\%$

FLTOVC05: Are there instances of functions not returning a value or throwing an exception on an error?

EXEMPLAR TOOL(S):

Mostly manual, but SA tools, such as clang-tidy or SonarQube, can help determine points of entry by finding any issues related to try/catch or unchecked return values. Samples of violations are collected throughout the analysis of the Fault Tolerance section by capturing violations while investigating other questions.

PROCESS:

Search across the code base for error patterns in methods, e.g., try/catch, error numbers, error logging. Find references to methods that have error pathways.

EVALUATION:

Calculate the number of files with problems (v) over the number of files reviewed (n), where $v / n = x$

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC06: Are pointers confirmed to be correctly initialized and non-null before use?

EXEMPLAR TOOL(S):

clang-tidy, CppCheck, SonarQube, etc.

PROCESS:

All quality static-analysis tools contain rules for uninitialized variables. This question encompasses both pointers and undefined behavior for uninitialized non-pointer values.

EVALUATION:

After manually confirming several cases, use the number of violations found by an analysis tool (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC07: Are memory allocations and deallocations matched?

EXEMPLAR TOOL(S):

clang-tidy, CppCheck, SonarQube

PROCESS:

Filter SA output violations by memory allocation and deallocation rules. Verify mismatches, such as allocating with “new” and freeing “free” or delete instead of delete[]. Additional rules may need to be filtered to find reallocation problems, such as freeing the original pointer whenever realloc fails.

EVALUATION:

After manually confirming several cases, use the number of violations found by an analysis tool (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Few	Medium High	$3\% \leq x < 4\%$
Some	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC08: Are memory allocations correctly handled in error blocks (try/catch in new, NULL in malloc)?

EXEMPLAR TOOL(S):

clang-tidy, CppCheck, SonarQube

PROCESS:

Filter SA output violations whenever malloc/calloc/realloc return values are not performed and for try/catch where errors are not thrown by value and caught by reference (potential memory leak).

EVALUATION:

After manually confirming several cases, use the number of violations found by an analysis tool (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$3\% \leq x$
Some	Medium	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC09: Are parameter values within functions confirmed to be within expected bounds before use?

EXEMPLAR TOOL(S):

Mostly manual, but tools can help determine points of entry to begin searching, such as comparing integers to floating-point values and error-prone object comparisons.

PROCESS:

Start with methods that have a lot of usage references, e.g., core API, as well as ones that have parameters. Determine whether the parameters have runtime checks. Analysts will need to also determine whether the callers perform checks prior to calling the API before evaluating.

EVALUATION:

In order to bound the scope of this question, at least 30 files that have more 5 or more functions should be checked for violations. Calculate the number of functions with violations (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC10: Are user inputs confirmed to be of the correct type?

EXEMPLAR TOOL(S):

Mostly manual, but static analysis tools, such as clang-tidy and SonarQube, can help determine points of entry to locate warnings related to slicing and non-explicit construction.

PROCESS:

Start with methods that have a lot of usage references, e.g., core API, as well as ones that have parameters. Determine whether the parameters have runtime checks. Analysts will need to also determine whether the callers perform checks prior to calling the API before evaluating.

EVALUATION:

In order to bound the scope of this question, at least 30 files that have more 5 or more functions should be checked for violations. Calculate the number of functions with violations (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOVC11: Are user inputs confirmed to be within an expected range?

EXEMPLAR TOOL(S):

Mostly manual, but tools can help determine points of entry to begin searching, such as finding potential integer overflow or negative bounds

PROCESS:

Start with methods that have a lot of usage references, e.g., core API, as well as ones that have parameters. Determine whether the parameters have runtime checks. Analysts will need to also determine whether the callers perform checks prior to calling the API before evaluating.

EVALUATION:

In order to bound the scope of this question, at least 30 files that have more 5 or more functions should be checked for violations. Calculate the number of functions with violations (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Mostly	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

10.3 Error Handling

FLTOEH01: Is there evidence of a systematic error handling mechanism?

EXEMPLAR TOOL(S):

Mostly manual, but tools have rules that may help, e.g., whenever try/catch patterns are flagged for not passing by value and catching by reference.

PROCESS:

Find consistency among error handling, answering the following sub-questions:

1. Determine whether any error handling exists in all parts of the code base

- Determine whether error-handling patterns are mixed throughout the same library/module/area of code

EVALUATION:

If error handling does not exist throughout the entire codebase, then the risk is high. If error-handling patterns are mixed, then the risk is medium. If the error-handling patterns are mixed within the same library/module, then the risk is medium-high.

SCORES:

CREW	Risk
No	High
Minimally	Medium High
Somewhat	Medium
Mostly	Medium Low
Yes	Low

FLTOEH02: Do functions / methods consistently return an error or throw an exception on failure?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Find whether there are pathways in a method/function that conceal that an error occurred (swallowing the error), fail to promote the occurrence of an error (failing to rethrow), or perform logging without providing information to the calling code.

EVALUATION:

In order to bound the scope of this question, at least 30 files that have more 5 or more functions should be checked for violations. Calculate the number of functions with violations (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Many	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOEH03: Is there evidence of a consistent error-handling pattern?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Look within modules/subsystems to verify whether each one consistently uses an error-handling pattern, such as try/catch, multi-value return, return integers. Different patterns between modules/subsystems are normal, but different patterns within a single module/subsystem are not. If no error handling occurs at all in certain modules, then that is evidence of a lack of an error-handling pattern.

EVALUATION:

In order to bound the scope of this question, at least 30 files that have more 5 or more functions should be checked consistency. Calculate the number of functions with violations (v) over the total number of functions (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$3\% \leq x$
Somewhat	Medium	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

10.4 Error Logging

FLTOEL01: Are errors consistently logged?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Based on the error-handling mechanisms in the abovementioned section, determine whether there is consistent logging of errors. Locate try/catch and error constants by searching the codebase.

EVALUATION:

Calculate the number of violations (v) over the number of files checked (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Mostly	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOEL02: Are exceptional conditions consistently logged?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Find whether try/catch or error codes are being logged. Locate try/catch and error constants by searching the codebase.

EVALUATION:

Calculate the number of violations (v) over the number of files checked (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$4\% \leq x$
Minimal	Medium High	$3\% \leq x < 4\%$
Somewhat	Medium	$2\% \leq x < 3\%$
Mostly	Medium Low	$1\% \leq x < 2\%$
Yes	Low	$0\% \leq x < 1\%$

FLTOEL03: Are security events consistently logged?**EXEMPLAR TOOL(S):**

Manual

PROCESS:

Find security-driven areas of code and determine whether all or some of the events are being logged. The events can be searched for by searching the available types of logging tags, e.g., critical, fatal. Search through files that are known to be security issues by searching for security keywords. It's possible this question is not applicable.

EVALUATION:

If security areas / files are not found, then disable this question. If security files are found, then select approximately 10 files to determine whether there are times when logging should occur but does not.

SCORES:

CREW	Risk	Description
No	High	7+ occurrences
Minimally	Medium High	5 – 6 occurrences
Somewhat	Medium	3 – 4 occurrences
Mostly	Medium Low	1 – 2 occurrences
Yes	Low	0 occurrences

FLTOEL04: Are logs sufficiently detailed to identify the issue?**EXEMPLAR TOOL(S):**

Manual

PROCESS:

Review log calls to determine what information they typically contain. Information should describe the problem, dump variables, class name, line numbers, error codes, and error messages. Search the codebase by the logging macros or logging libraries used.

EVALUATION:

Select 30 logging statements in different parts of the codebase. Calculate the number of logging statements that are not sufficiently detailed to the total number of statements manually checked. If there are not 30 logging statements, then the risk is high.

SCORES:

CREW	Risk	Description
No	High	$13.3 \leq x$
Minimally	Medium High	$10\% \leq x < 13.3\%$
Somewhat	Medium	$6.67\% \leq x < 10\%$
Mostly	Medium Low	$3.3\% \leq x < 6.67\%$
Yes	Low	$0\% \leq x < 3.3\%$

FLTOEL05: Are stored logs maintained to not exceed a maximum allowable size?**EXEMPLAR TOOL(S):**

Manual

PROCESS:

Locate how logs get stored, if at all. There are typically multiple loggers, so this can be an iterative process. Additionally, check for event servers and middleware software that provides logging mechanisms. Find configurations related to storage sizes.

EVALUATION:

If loggers do not log to file, then this is high risk. If loggers do not have a cap or check for sizes, then this is high risk on embedded systems, when not on an embedded system then the risk is medium (due to large amounts of disk space readily available in non-embedded systems). If loggers ship to a log server but do not check for sizes, then the risk is medium. If there are mixed cases where some loggers check for sizes and others do not, then the average is taken where a risk assessment is scored between 1 – 5, where 5 is the highest.

SCORES:

CREW	Risk
No	High
Minimally	Medium High
Somewhat	Medium
Mostly	Medium Low
Yes	Low

10.5 Instrumentation**FLTOIN01: Is there evidence of instrumentation at all?****EXEMPLAR TOOL(S):**

Manual

PROCESS:

Locate diagnostics, performance, metrics, utilization, or events within the code base. Determine whether ANY instrumentation exists at all. The type of instrumentation may exist outside of the subsequent questions in this section.

EVALUATION:

Risk is based on the amount of information captured related to processor utilization.

SCORES:

CREW	Risk	Description
No	High	No evidence
Yes	Low	Evidence

FLTOIN02: Are worst-case and average run-time processor utilization recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Locate diagnostics, performance, metrics, utilization, or events within the code base. Determine what metrics, if any, are being tracked and how. This can include periodic updates. Determine what kind of statistics are measured and how those statistics are tracked.

EVALUATION:

Risk is based on the amount of information captured related to processor utilization.

SCORES:

CREW	Risk	Description
No	High	no information
Minimally	Medium High	basic processor information, but not utilization
Somewhat	Medium	real-time or on-demand utilization recorded
Mostly	Medium Low	average or worst-case utilization recorded
Yes	Low	average and worst-case utilization recorded

FLTOIN03: Are run-time communication channels utilization recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Check for COTS middleware to see whether any out-of-box solutions are configured for performance information or policies. Check for custom communication modules and determine what metrics are included.

EVALUATION:

In practice, this is typically a binary choice. Either the utilization exists or it doesn't.

SCORES:

CREW	Risk	Description
No	High	No
Minimally	Medium High	n/a
Somewhat	Medium	n/a
Mostly	Medium Low	n/a
Yes	Low	Yes

FLTOIN04: Are worst-case and average run-time memory utilization recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Search for OS-specific diagnostics. If any exist, then determine what information is available related to memory.

EVALUATION:

Risk is based on the amount of information captured related to processor utilization.

SCORES:

CREW	Risk	Description
No	High	No information
Minimally	Medium High	Basic memory information, but not utilization
Somewhat	Medium	Real-time or on-demand utilization recorded
Mostly	Medium Low	Average or worst-case utilization recorded
Yes	Low	Average and worst-case utilization recorded

FLTOIN05: Are worst-case and average run-time timing latencies recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Find communication latencies and performance-enforced latencies. Determine how latencies are tracked.

EVALUATION:

Risk is based on the amount of information captured related to processor utilization.

SCORES:

CREW	Risk	Description
------	------	-------------

No	High	no information
Minimally	Medium High	basic memory information, but not utilization
Somewhat	Medium	real-time or on-demand utilization recorded
Mostly	Medium Low	average or worst-case utilization recorded
Yes	Low	average and worst-case utilization recorded

FLTOIN06: Are boot-up (turn on) counts recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Find code that targets embedded devices, custom boards, and then find anything related to the operating system. Locate anything related to device reboots and determine what information is tracked.

EVALUATION:

In practice, this is typically a binary choice. Either some form of boot-up records exists or they don't.

SCORES:

CREW	Risk	Description
No	High	No
Minimally	Medium High	n/a
Somewhat	Medium	n/a
Mostly	Medium Low	n/a
Yes	Low	Yes

FLTOIN07: Are system (total) run-time hours recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Find code that targets embedded devices, custom boards, and then find anything related to the operating system. Locate anything related to runtime and determine what information is tracked.

EVALUATION:

If runtime is not accounted for then the risk is high. If there are multiple devices and some of them contain runtime while others don't then the risk is medium.

SCORES:

CREW	Risk
No	High
Minimally	Medium High

Somewhat	Medium
Mostly	Medium Low
Yes	Low

10.6 Health Monitoring

FLTOHM01: Does the system utilize a watchdog timer?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Find code that targets embedded devices, custom boards, and middleware frameworks. Locate anything related to watchdog or task health managers.

EVALUATION:

In practice, this is typically a binary choice, although there are circumstances where multiple watchdogs can or should exist. In mixed cases, the risk is medium.

SCORES:

CREW	Risk
No	High
Minimally	Medium High
Somewhat	Medium
Mostly	Medium Low
Yes	Low

FLTOHM02: Does the code verify the health of all tasks and threads before strobing the watchdog?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Locate system and health monitoring for operating system and all devices. Additionally, locate thread pools or thread management. Determine how tasks and threads are monitored, and then find out whether any management of threads occurs before being killed or restarted.

EVALUATION:

Health verification either occurs or it doesn't.

SCORES:

CREW	Risk
No	High
Yes	Low

FLTOHM03: Are fault and watchdog reboot counts recorded?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Locate system and health monitoring for the operating system and all devices. Determine whether reboots are monitored, including thread and task restarts.

EVALUATION:

Counts either exist or they don't.

SCORES:

CREW	Risk
No	High
Yes	Low

FLTOHM04: Are system health and instrumentation data accessible to system administrators?

EXEMPLAR TOOL(S):

Manual

PROCESS:

Locate system and health monitoring for operating system and all devices. Determine how the events and monitoring data are recorded. If recording occurs, then find out where the storage system is, such as a log server, event store. Determine whether a sys admin would be able to access the data.

EVALUATION:

If system health data is not logged, then the risk high. If data is logged and recorded to disk / database, then the risk will invariably be low. If data is not recorded anywhere, but exists in non-permanent storage, such as RAM, then the risk is high.

SCORES:

CREW	Risk	Description
No	High	No
Minimally	Medium High	n/a
Somewhat	Medium	n/a
Mostly	Medium Low	n/a
Yes	Low	Yes

11 Testability

11.1 Overview

This tab captures the risk factors associated with the testability of the system. Testing is one of the most consistently overlooked and underutilized methods of reducing defects, preventing side-effects, and establishing confidence in a product. Testability is the degree to which a system or component facilitates the establishment of test criteria, and the performance of tests to determine whether those criteria have been met [IEEE 1990]. Essentially, the testability of a program P is the probability that if P contains faults, P will fail under test [ISSTA 1993]. Testable code should be highly modular, self-contained, and simple. It should have a singular and well-defined purpose, few external dependencies, and few decision paths.

To receive a satisfactory score for this tab, there should be evidence of unit testing, especially automated testing (test cases, test harness, etc.). The code should utilize a test harness, test cases, and unit testing that provides ample code coverage and goes beyond happy paths to test edge cases. There should be evidence of regression testing, preferably automated regression testing. Also, there should be a low cyclomatic complexity and path count to allow thorough testing. The Testability tab is shown in Figure 11.1.1.

Figure 11.1.1 The Testability Tab

Focus	W	ID	I	Measurement / Question	A	Finding	D
UNIT	2.7	0.350	TS_UN01	1 Does the supplied code base include the unit test code?		No	0
			TS_UN02	1 Do unit tests use meaningful test data?		Some	0
			TS_UN02	1 Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?		Few	0
			TS_UN03	1 Do unit tests appear designed to test edge cases, i.e. go beyond happy-path testing?		Some	0
			TS_UN04	1 Do unit tests utilize assertions liberally to indicate test success or failure?		Many	0
			TS_UN05	1 Does the coverage of the unit tests appear to be comprehensive?		Yes	0
			TS_UN06	1 Does the code base utilize global variables?		No	0
			TS_UN07	1 Does the test code allow easy documentation of application or library code?		Some	0
			TS_UN08	1 Do tests utilize both positive and negative test cases, i.e. fault injection?		Yes	0
REGRESSION	3.0	0.300	TS_UN09	1 Are the unit tests generally easy to understand and verifiable?		No	0
			TS_RG01	1 Is there evidence in either code or comments that the code is regression tested?		Some	0
			TS_RG02	1 Is there evidence of test driver usage or other automated regression testing?		No	0
COMPLEXITY	2.6	0.350	TS_RG03	1 Is there evidence of functional testing, i.e. testing slices of functionality within the whole system?		Yes	0
			TS_CM01	1 Maximum cyclomatic complexity (CC1) for any function within the code base		• 151	0
			TS_CM02	0 Total number of functions or methods within the code base		• 8157	0
			TS_CM03	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10		97.8%	0
			TS_CM04	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and <= 20		1.5%	0
			TS_CM05	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and <= 30		0.4%	0
			TS_CM06	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and <= 40		0.1%	0
			TS_CM07	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and <= 50		0.0%	0
			TS_CM08	1 Number of functions within the code with cyclomatic complexity (CC1) > 50		8	0
			TS_CM09	1 Average file cyclomatic complexity (CC1)		• 2.58	0
			TS_CM10	1 Average file strict cyclomatic complexity (CC2)		2.75	0
			TS_CM11	1 Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10		2.6%	0
			TS_CM12	1 Average file modified cyclomatic complexity (CC3)		2.49	0
			TS_CM13	1 Percentage of functions or methods within the code with Hersteller PATH exceeding 80		1.9%	0
RISK	2.73						

The following sections describe how to complete the CREW Testability tab:

11.2 Unit

TS_UN01: Does the supplied code base include the unit test code?

TOOL(S):

Manual

PROCESS:

Determine whether any unit tests are included in the code delivered.

EVALUATION:

The evaluation of unit tests is reviewed only in relation to the module they exist within. For example, if there is a networking module and a database module, but unit tests only exist within the networking module, then only some unit tests are provided.

SCORES:

CREW	Risk	Description
No	High	< 40% of the modules do not contain unit tests
Few	Medium High	40% of the modules contain unit tests
Some	Medium	60% of the modules contain unit tests
Many	Medium Low	80% of the modules contain unit tests
Yes	Low	All modules contain unit tests

TS_UN02: Do unit tests use meaningful test data?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether the data used is meaningful, i.e., does the data reasonably fit the description of the test or is the data contrived for the purposes of making the test succeed or fail only?

EVALUATION:

Calculate the number of test files that contain non-meaningful data (v) over the total number of test files (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$20\% \leq x$
Few	Medium High	$15\% \leq x < 20\%$
Some	Medium	$10\% \leq x < 15\%$
Many	Medium Low	$5\% \leq x < 10\%$
Fully	Low	$0\% \leq x < 5\%$

TS__UN02: Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether test harnesses, mocked dependencies and stubbed methods are provided.

EVALUATION:

Determine whether harnesses and mocking are provided for all available modules. This does not account for the 0% where no unit tests are provided. This only investigates unit tests that do exist.

SCORES:

CREW	Risk
No	High
Minimal	Medium High
Somewhat	Medium
Many	Medium Low
Yes	Low

TS__UN03: Do unit tests appear designed to test edge cases, e.g. go beyond happy-path testing?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether negative, positive, exceptions, and errors are accounted for.

EVALUATION:

Calculate the number of test files that contain multiple assertions/tests that perform more than simple checks, where only the expected outcome is verified, (v) over the total number of test files (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$20\% \leq x$
Few	Medium High	$15\% \leq x < 20\%$
Some	Medium	$10\% \leq x < 15\%$
Many	Medium Low	$5\% \leq x < 10\%$
Fully	Low	$0\% \leq x < 5\%$

TS__UN04: Do unit tests utilize assertions liberally to indicate test success or failure?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether assertions are provided for more than one possible outcome.

EVALUATION:

Calculate the number of test files that contain multiple assertions/tests (v) over the total number of test files (n), where $v / n = x$.

SCORES:

CREW	Risk	Description
No	High	$20\% \leq x$
Few	Medium High	$15\% \leq x < 20\%$
Some	Medium	$10\% \leq x < 15\%$
Many	Medium Low	$5\% \leq x < 10\%$
Fully	Low	$0\% \leq x < 5\%$

TS__UN05: Does the coverage of the unit tests appear to be comprehensive?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether unit tests attempt to cover as much of the code as delivered.

EVALUATION:

If there are test drivers or scripts, then determine whether percentages are calculated for code coverage. If no drivers / scripts / percentages are provided, then manually estimate the approximate coverage by checking the unit test files. If there are too many files to manually verify, then select several files to get a sense of coverage on average. $x = \text{est. coverage per file} / \text{test files}$

SCORES:

CREW	Risk	Description
No	High	$20\% \leq x$
Few	Medium High	$15\% \leq x < 20\%$
Some	Medium	$10\% \leq x < 15\%$
Many	Medium Low	$5\% \leq x < 10\%$
Fully	Low	$0\% \leq x < 5\%$

TS__UN06: Does the code base utilize global variables?

TOOL(S):

clang-tidy, SonarQube, etc.

PROCESS:

Determine whether any global variables exist in the code base at all. Globals that are encapsulated in namespaces or are provided as public static const members in a class are not considered risks, unless they are not self-documented, e.g., “static const int m = 4;”.

EVALUATION:

Evaluation in practice consists of three options defined in the scores table below.

SCORES:

CREW	Risk	Description
No	High	Globals exist in multiple files or extensively in a single file
Few	Medium High	<i>n/a</i>
Somewhat	Medium	Global exist
Many	Medium Low	<i>n/a</i>
Fully	Low	No

TS__UN07: Does the test code allow easy documentation of application or library code?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether the test code is easy to follow and provides adequate comments indicating what’s being tested.

EVALUATION:

If unit tests aren’t provided or the tests are sparse then the risk is high. If there are only positive test cases then the risk is medium. Otherwise, the risk is low.

SCORES:

CREW	Risk
No	High
Somewhat	Medium
Fully	Low

TS__UN08: Do tests utilize both positive and negative test cases, e.g. fault injection?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then check whether the tests force exceptions, errors, and unexcepted conditions.

EVALUATION:

In practice, this is typically one of three options outlined in the table below. If there are no unit tests then the risk is high.

SCORES:

CREW	Risk	Description
No	High	no forced expectations or injected errors
Rarely	Medium High	n/a
Somewhat	Medium	some forced exceptions or injected errors
Frequently	Medium Low	n/a
Yes	Low	most of the tests have forced exceptions or errors

TS__UN09: Are the unit tests generally easy to understand and verifiable?

TOOL(S):

Manual

PROCESS:

If unit tests are provided, then determine whether the test code performs useful tests that exercise the code and that the code is not overly complex.

EVALUATION:

If there are no unit tests, then the risk is high. If tests typically cover multiple units, then the risk is high. If there is a mix of isolated units and multiple units being tested, then the risk is medium. Otherwise, it's low.

SCORES:

CREW	Risk
No	High
Somewhat	Medium
Fully	Low

11.3 Regression

TS__RG01: Is there evidence in either code or comments that the code is regression tested?

TOOL(S):

Manual

PROCESS:

Search for any evidence of regression tests. Evidence is based on comments within the code and any materials provided to the SEI, such as scripts and documentation. If tests are included, then determine how comprehensive those regressions tests are.

EVALUATION:

In practice, this is a binary question.

SCORES:

CREW	Risk	Descriptions
No	High	No evidence
Some	Medium	<i>n/a</i>
Fully	Low	Comments or code indicate tests exist

TS_RG02: Is there evidence of test driver usage or other automated regression testing?

TOOL(S):

Manual

PROCESS:

If regression tests are available, then determine mechanism for driving the tests, such as scripts.

EVALUATION:

Evaluation is outlined in the table below.

SCORES:

CREW	Risk	Description
No	High	No evidence
Some	Medium	indirect evidence found in comments or documents
Fully	Low	direct evidence of drivers / scripts provided

TS_RG03: Is there evidence of functional testing, e.g. testing slices of functionality within the whole system?

TOOL(S):

Manual

PROCESS:

Search for functional testing, such as end-to-end tests, integration tests, etc. If they are available, determine how comprehensive the tests are.

EVALUATION:

In practice, this primarily reviews end-to-end tests or drivers that test several areas of code at a time. The quality of the tests is reviewed for expected and unexpected outcomes, detail of output, quantity of test data available, and how much area of the codebase is touched by the tests.

SCORES:

CREW	Risk	Description
No	High	no tests available or only positive testing
Some	Medium	positive and negative tests with some tests available
Fully	Low	comprehensive coverage with positive and negative tests

11.4 Complexity

Code complexity is a prime consideration for testability as well as maintainability, however CREW also considers code complication as well. So, what's the difference? Let's clarify their meaning as utilized within CREW.

Complexity

The total cyclomatic complexity (CC) of an application is largely a function of the problem it solves. The important consideration is separating the total CC into small enough "chunks" (i.e. low enough function CC) so each can be fully unit tested, e.g. to obtain basis path test coverage. Hence, code complexity measures tie closely with testability.

Complication

Complication in code is more a function of the code's design and implementation decisions beyond just function CC. This is apparent by the code's language scope metric (VOCF, a function of the number of operators & operands used) which is a Hersteller (HIS) measure of code maintenance difficulty, the depth of control structure nesting (switch, if / else, do / while), function Fanin (called by) and Fanout (calling), the number of parameters functions take, use of recursion, etc. Hence, code complication measures tie closely with maintainability.

For the Testability | Complexity focus, we compute a CREW score for the codebase complexity metrics with emphasis on cyclomatic complexity. Use any SA tool to determine these metrics, however here, for the SA tool Understand (Scientific Toolworks), we also include the metric's API or CodeCheck, shown in parentheses.

TS__CM01: Maximum cyclomatic complexity (CC1) for any function within the code base

Auto-populated from System tab, Metrics Focus, Item #18.

TS__CM02: Total number of functions or methods within the code base

Auto-populated from System tab, Metrics Focus, Item #16.

TS__CM03: Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10

Run CodeCheck (rule ID METRIC_13) using standard cyclomatic complexity (CC1) and a maximum limit of 10 to obtain the number of functions (or methods) within the code base exceeding

this limit. Note this $CC1 > 10$ value for later use. Subtract the $CC1 > 10$ value from the total number of functions in the code base (from System tab, Metrics Focus, Item #16) then divide this difference by the total number of functions in the code base and enter the result as a percentage.

TS__CM04: Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and ≤ 20

Run CodeCheck (rule ID METRIC_13) again using standard cyclomatic complexity (CC1) and a maximum limit of 20 to obtain the number of functions (or methods) within the code base exceeding this limit. Note this $CC1 > 20$ value for later use and subtract it from the $CC1 > 10$ value then divide this difference by the total number of functions in the code base and enter the result as a percentage.

TS__CM05: Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and ≤ 30

Similarly, run CodeCheck (rule ID METRIC_13) again using standard cyclomatic complexity (CC1) and a maximum limit of 30 to obtain the number of functions (or methods) within the code base exceeding this limit. Note this $CC1 > 30$ value for later use and subtract it from the $CC1 > 20$ value then divide this difference by the total number of functions in the code base and enter the result as a percentage.

TS__CM06: Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and ≤ 40

Again, run CodeCheck (rule ID METRIC_13) using standard cyclomatic complexity (CC1) and a maximum limit of 40 to obtain the number of functions (or methods) within the code base exceeding this limit. Note this $CC1 > 40$ value for later use and subtract it from the $CC1 > 30$ value then divide this difference by the total number of functions in the code base and enter the result as a percentage.

TS__CM07: Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and ≤ 50

Again, run CodeCheck (rule ID METRIC_13) using standard cyclomatic complexity (CC1) and a maximum limit of 50 to obtain the number of functions (or methods) within the code base exceeding this limit. Note this $CC1 > 50$ value for later use and subtract it from the $CC1 > 40$ value then divide this difference by the total number of functions in the code base and enter the result as a percentage.

TS__CM08: Number of functions within the code with cyclomatic complexity (CC1) > 50

Enter the $CC1 > 50$ value as a simple violation count.

TS__CM09: Average file cyclomatic complexity (CC1)

[Auto-populated from System tab, Metrics Focus, Item #17.](#)

TS__CM10: Average file strict cyclomatic complexity (CC2)

Enter the average file strict cyclomatic complexity for the code (AvgCyclomaticStrict).

TS__CM11: Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10

Run CodeCheck (rule ID METRIC_13) using strict cyclomatic complexity (CC2) and a maximum limit of 10 to obtain the number of functions (or methods) within the code base exceeding this limit. Divide CC2 > 10 value by the total number of functions in the code base and enter the result as a percentage.

TS__CM12: Average file modified cyclomatic complexity (CC3)

Enter the average file modified cyclomatic complexity for the code (AvgCyclomaticModified).

TS__CM13: Percentage of functions or methods within the code with Hersteller PATH exceeding 80

Run CodeCheck (rule ID HIS_02) for the Hersteller Initiative Software (HIS) PATH metric using the HIS limit of 80 to obtain the number of functions (or methods) within the code base exceeding this limit then divide this value by the total number of functions in the code base and enter the result as a percentage.

12 Security General

12.1 Overview

Security of code consists of techniques to protect a system from known or novel vulnerabilities so that it continues to function as intended and remains safe from attacks. Secure code is exemplified by adherence to secure coding standards such as the SEI CERT Secure Coding standards. More generally, secure code embodies accepted best practices regarding defensive coding, validation, encryption, libraries, API's, and the traceability of executable and data files.

To receive a satisfactory score for this tab, a codebase should have a high level of defensive, secure coding practices. It should validate untrusted inputs/interfaces and minimize the use of potentially unsafe library calls. There should be evidence that code compiles with no (or few) warnings, and evidence of the effective use of executable and parameter file signatures/anti-tamper measures.

When performing this section of the CREW analysis it is important to note that some questions, predominantly in the Defensiveness and Libraries foci, can be answered very simply with SA tools. However, most questions rely more heavily on manual code review. It is recommended analysts start by finding evidence of encryption. Once the functions used for encryption are identified, the analyst can use SA tools or an IDE to find instances of the function that help answer the CREW questions.

Scoring

Many questions in the Security General section are answered with a drop-down selection. As a general guideline, here are the standard thresholds for which selection to choose for most questions in the Security General tab. Questions that deviate from these thresholds will list their own threshold table.

Selection	Sampled Elements with Violation
All/Yes	None or a small handful
Many/Most	< 25%
Somewhat	< 50%
Few/Seldom	< 80%
No	80% or higher

Note that these are guidelines and scores, particularly near boundary points, must be judged by the analyst. The Security General tab is shown in Figure 12.1.1.

Figure 12.1.1 The Security General Tab

Focus	Risk	W	ID	I	Measurement / Question	A	Finding	D	Note
DEFENSIVENESS	2.6	0.200	SEGNDE01	1	Are magic numbers excluding the values (+ and -) 0, 0.0, 1, 1.0, 2, 4, 8, 10, 16 prevalent in the code?		No	0	
			SEGNDE02	1	Are expressions within macros fully parenthesized?		Few	0	
			SEGNDE03	1	Are numbers defined by the preprocessor parenthesized?		Some	0	
			SEGNDE04	1	Are macros reserved for simple highly reusable functions?		Many	0	
			SEGNDE05	1	Are variables checked before used in operations that may cause overflows or divide by zero errors?		Yes	0	
			SEGNDE06	1	Are pointers confirmed to be correctly initialized and non-null before use?		No	0	
			SEGNDE07	1	Are parameters within functions confirmed to be within expected bounds before use?		Few	0	
			SEGNDE08	1	Are return values checked for errors?		Some	0	
			SEGNDE09	1	Is there evidence of a unifying coding standard?		Many	0	
			SEGNDE10	1	Are exceptional conditions handled consistently?		Yes	0	
			SEGNDE11	1	Does the code exhibit graceful degradation?		No	0	
			SEGNDE12	1	Is trusted and untrusted data stored within the same data structure?		Few	0	
VALIDATION	2.9	0.200	SEGNVA01	1	Is hashing used to verify data sources?		No	0	
			SEGNVA02	1	Are modern hashing algorithms used?		Yes	0	
			SEGNVA03	1	Are encryption algorithms used when passing data through untrusted channels?		No	0	
			SEGNVA04	1	Is sensitive data at rest encrypted?		Few	0	
			SEGNVA05	1	Are modern encryption algorithms used?		Some	0	
			SEGNVA06	1	Is memory cleaned between reads?		Many	0	
			SEGNVA07	1	Is message acknowledgment utilized?		Yes	0	
			SEGNVA08	1	Are interrupts used over polling for communication buffers?		No	0	
			SEGNVA09	1	Does privileged activity undergo secondary authentication?		Few	0	
			SEGNVA10	1	Is all user input confirmed to be of the correct type?		Some	0	
			SEGNVA11	1	Is all user input confirmed to be within an expected range?		Many	0	
			SEGNVA12	1	Is all user input sanitized?		Yes	0	
LIBRARIES	2.7	0.200	SEGNLI01	1	Is a real-time OS used?		No	0	
			SEGNLI02	1	Are any library calls or run-time API's with known security concerns utilized?		Few	0	
			SEGNLI03	1	Are any deprecated library functions used?		Some	0	
			SEGNLI04	1	Are returned values from library functions consistently checked and verified?		Many	0	
			SEGNLI05	0	Does the code base utilize any 3rd-party source code?		Yes	0	
			SEGNLI06	1	Are the most recent versions of third-party source code used?	C	No	0	
			SEGNLI07	1	Is there evidence that third-party source code has been modified for system utilization?	C	No	0	
			SEGNLI08	1	Do all third-party assets originate from secure sources?		No	0	
WARNINGS	3.6	0.200	SEGNWA01	1	Is there evidence that the code complies with zero (0) warnings?		No	0	
			SEGNWA02	1	Was the code compiled as part of this code risk estimation analysis?		Modified	0	
			SEGNWA03	1	Does the supplied makefile set warning flags to their highest level, e.g. -Wall, -Wextra, -Weverything?		Yes	0	
			SEGNWA04	1	Warnings count if the code is compiled with the highest warning flags set? (enter 25 if not compiled)		250	0	
			SEGNWA05	1	Are warning suppressions present within the code base?		Yes	0	
TRACEABILITY	2.9	0.200	SEGNTR01	1	Does the code validate the executable image against a digital signature to ensure anti-tamper?		No	0	
			SEGNTR02	1	Do parameter files used by the program contain a signature that is verified at boot-time?		Some	0	
			SEGNTR03	1	Are parameter files used by the program encrypted?		Yes	0	
RISK	2.93								

12.2Defensiveness

SEGNDE01: Are magic numbers excluding the values (+ and -) 0, 0.0, 1, 1.0, 2, 4, 8, 10, 16 prevalent in the code?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to check for instances of magic numbers.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	5+ per ksloc
Many/Most	< 5 per ksloc
Somewhat	< 2.5 per ksloc
Few/Seldom	< 0.5 per ksloc

No	None or a small handful
----	-------------------------

SEGNDE02: Are expressions within macros fully parenthesized?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for un-parenthesized macros expressions.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None
Many/Most	< 5
Somewhat	< 10
Few/Seldom	< 20
No	20+

SEGNDE03: Are numbers defined by the preprocessor parenthesized?

EXEMPLAR TOOL(S): grep

PROCESS: Search the project for ‘#define’.

EVALUATION: Ensure all numbers are parenthesized.

SCORING: Use standard thresholds

SEGNDE04: Are macros reserved for simple highly reusable functions?

EXEMPLAR TOOL(S): Understand

PROCESS: Use a static analyzer to search for function-like macros.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None or a small handful
Many/Most	< 10
Somewhat	< 50
Few/Seldom	< 100
No	100+

SEGNDE05: Are variables checked before being used in operations that may cause overflows or divide by zero errors?

EXEMPLAR TOOL(S): Regex search

PROCESS: Use regex to search for cases of division, addition, and subtraction.

EVALUATION: Ensure the prevention of any divide by zero errors. Similarly, cases of addition and subtraction should be analyzed to ensure the impossibility of overflows.

SCORING:

Selection	Number of Violations Found
All/Yes	None
Many/Most	< 5
Somewhat	< 10
Few/Seldom	< 20
No	20+

SEGNDE06: Are pointers confirmed to be correctly initialized and non-null before use?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample files.

EVALUATION: Ensure all pointers are properly checked before use.

SCORING: Use standard thresholds

SEGNDE07: Are parameters within functions confirmed to be within expected bounds before use?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample functions.

EVALUATION: Ensure functions confirm their parameters are within bounds.

SCORING: Use standard thresholds

SEGNDE08: Are function return values checked for errors?

EXEMPLAR TOOL(S): Understand

PROCESS: Use SA to check for unused return values.

EVALUATION: Verify instances are true positives.

SCORING:

Selection	Number of Violations Found
All/Yes	None or a small handful
Many/Most	< 0.5 per ksloc
Somewhat	< 2.5 per ksloc
Few/Seldom	< 5 per ksloc
No	5+ per ksloc

SEGNDE09: Is there evidence of a unifying coding standard?

EXEMPLAR TOOL(S): CREW

PROCESS/EVALUATION: Refer to the Stylistic Standards focus.

SCORING:

Selection	Stylistic Standards Score
Yes	< 1.5
Mostly	< 2
Somewhat	< 3
Minimal	< 4
No	4+

SEGNDE10: Are exceptional conditions handled consistently?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Sample functions.

EVALUATION: Ensure that functions have contingencies in place for any possible exceptions.

SCORING: Use standard thresholds

SEGNDE11: Does the code exhibit graceful degradation?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: It is recommended the analyst not answer this question until the end of the analysis, since the more code that is analyzed, the better sense an analyst will have of how to answer the question. Searching for phrases like ‘grace’ or ‘recover’ may sometimes help, but seeing as much of the code as possible is the best way to answer this question.

EVALUATION: For this question, examine any errors and exceptions and ensure contingencies are in place to handle and recover from any errors or exceptions.

SCORING: Use standard thresholds

SEGNDE12: Is trusted and untrusted data stored within the same data structure?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Find any data structures holding trusted data.

EVALUATION: Ensure untrusted data is not also included in the structure.

SCORING: Yes or No

12.3 Validation

SEGNVA01: Is hashing used to verify data sources?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for “hash” and confirm usage.

EVALUATION: Verify instances are true positives.

SCORING: Use standard thresholds

SEGNVA02: Are modern hashing algorithms used?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for “hash” and confirm usage.

EVALUATION: Ensure that any hashes used for cryptographic functions adhere to the proper government standards.

EVALUATION: Yes or No

SEGNVA03: Are encryption algorithms used when passing data through untrusted channels?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for phrases such as “crypt”, “encode”, and “decode” to find evidence of encryption.

EVALUATION: Make sure encryption functions are in use for all message classes or functions.

SCORING: Yes or No

SEGNVA04: Is sensitive data at rest encrypted?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for phrases such as “crypt”, “encode”, and “decode”.

EVALUATION: Make sure encryption functions are in use on all data containers or I/O files.

SCORING: Use standard thresholds

SEGNVA05: Are modern encryption algorithms used?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for phrases such as “crypt”, “encode”, and “decode”.

EVALUATION: Make sure encryption algorithms used are modern and meet relevant government standards.

SCORING: Yes or No

SEGNVA06: Is memory cleaned between reads?

EXEMPLAR TOOL(S): grep

PROCESS: Searching for terms like “memset”, “clear”, “clean”, “buf”, and “read” to find cases of message buffers.

EVALUATION: Ensure message buffers are cleaned between read.

SCORING: Use standard thresholds

SEGNVA07: Is message acknowledgment utilized?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for “ack” and “acknowledge”. You may also examine any functions or classes pertaining to messages or communications.

EVALUATION: Ensure all utilize some form of message acknowledgement.

SCORING: Use standard thresholds

SEGNVA08: Are interrupts used over polling for communication buffers?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for “poll” to find any evidence of polling being used.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

SEGNVA09: Does privileged activity undergo secondary authentication?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for phrases such as “authenticate”, “validate”, “password”, “privilege”, “verify”, and “adm”.

EVALUATION: Ensure multiple forms of authentication are used for any found privileged activity.

SCORING: Yes or No

SEGNVA10: Is all user input confirmed to be of the correct type?

EXEMPLAR TOOL(S): grep

PROCESS: Search for any functions that are reading in text from the keyboard or from text files.

EVALUATION: Ensure all such input is ensured to be of the expected type.

SCORING: Use standard thresholds

SEGNVA11: Is all user input confirmed to be within an expected range?

EXEMPLAR TOOL(S): grep

PROCESS: Search for any functions that are reading in text from the keyboard or from text files.

EVALUATION: Ensure all such input is checked to be in expected bounds.

SCORING: Use standard thresholds

SEGNVA12: Is all user input sanitized?

EXEMPLAR TOOL(S): grep

PROCESS: Search for any functions that are reading in text from the keyboard or from text files.

EVALUATION: Ensure all such input is sanitized.

SCORING: Use standard thresholds

12.4 Libraries

SEGNLI01: Is a real-time OS used?

EXEMPLAR TOOL(S): CREW

PROCESS: Refer to the System tab to see OS being used.

EVALUATION: Ensure the OS used is an RTOS.

SCORING: Yes or No

SEGNLI02: Are any library calls or run-time API's with known security concerns utilized?

EXEMPLAR TOOL(S): Understand, grep

PROCESS: Use static analysis tools or search project files for uses of any unsafe function or library calls.

EVALUATION: Resources such as CERT Secure Coding Rules, MITRE's Common Weakness Enumerations, MISRA, or AUTOSARS can be useful in identifying any libraries or API's with known security concerns.

SCORING: Yes or No

SEGNLI03: Are any deprecated library functions used?

EXEMPLAR TOOL(S): Understand, grep

PROCESS: Use static analysis tools or search project files for uses of any unsafe function or library calls.

EVALUATION: Resources such as CERT Secure Coding Rules, MITRE's Common Weakness Enumerations, MISRA, or AUTOSARS can be useful in identifying any libraries or API's with known security concerns.

SCORING: Yes or No

SEGNLI04: Are returned values from library functions consistently checked and verified?

EXEMPLAR TOOL(S): Understand

PROCESS: Use static analysis tools to search for unused return values.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

SEGNLI05: Does the code base utilize any 3rd-party source code?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Search code base for any folders containing 3rd party code.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

SEGNLI06: Are the most recent versions of third-party source code used?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Check version numbers of any found 3rd party code

EVALUATION: Ensure it is the most recent version.

EVALUATION: Yes or No

SEGNLI07: Is there evidence that third-party source code has been modified for system utilization?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Examine 3rd party source code and, if possible, compare against a fresh copy of the code to see if any modifications have been made.

SCORING: Yes or No

SEGNLI08: Do all third-party assets originate from secure and reputable sources?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Examine the codebase and compile a list of all third-party assets used. Ensure these assets originate from a reputable source suitable for government use.

SCORING: Yes or No

12.5 Warnings

SEGNWA01: Is there evidence that the code compiles with zero (0) warnings?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Check build output for warnings.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

SEGNWA02: Was the code compiled as part of this code risk estimation analysis?

Self explanatory.

EVALUATION: Yes or No

SEGNWA03: Does the supplied makefile set warning flags to their highest level, e.g. -Wall, -Wextra, -Weverything?

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Ensure supplied build files include the relevant tags. Keep in mind that different compilers will have different flags for setting warning to the highest level.

EVALUATION: Check the documentation for the used compiler to determine the appropriate flags.

SCORING: Yes or No

SEGNWA04: Warnings count if the code is compiled with the highest warning flags set? (enter 25 if not compiled)

EXEMPLAR TOOL(S): Manual inspection.

PROCESS: Check build output for warnings.

EVALUATION: Verify instances are true positives.

SCORING: Input number of warnings found in build output, or 25 if no compilation occurred.

SEGNWA05: Are warning suppressions present within the code base?

EXEMPLAR TOOL(S): Understand, grep

PROCESS: Search project files for “#pragma” to find any warning suppressions.

EVALUATION: Verify instances are true positives.

SCORING: Yes or No

12.6 Traceability

SEGNTTR01: Does the code validate the executable image against a digital signature to ensure anti-tamper?

EXEMPLAR TOOL(S): grep

PROCESS: Search project files for phrases such as “tamper” and “signature.”

EVALUATION: Check if a digital signature of the executable is used.

SCORING: Yes or No

SEGNTTR01: Do parameter files used by the program contain a signature that is verified at boot-time?

EXEMPLAR TOOL(S): grep

PROCESS: Search project files for phrases such as “signature” and “verify”. Also search for “boot” to search for boot-time operations.

EVALUATION: Check for validating parameter files.

SCORING: Yes or No

SEGNTTR01: Are parameter files used by the program encrypted?

EXEMPLAR TOOL(S): grep

PROCESS: Search project for phrases such as “crypt”, “encode”, and “decode”. Searching for relevant file opening/reading functions can also be helpful to find relevant files.

EVALUATION: Ensure encryption functions are used on any relevant files being read in.

SCORING: Use standard thresholds

13 Security CERT

13.1 Overview

This tab captures how the amount and types of CERT Secure Coding Rule violations impact the risk present in the code. The CERT C and CERT C++ secure coding standards were developed by SEI CERT to develop safe, reliable, and secure systems. To receive a satisfactory score in this tab, the code base should have both a low density of secure coding issues and a low number of high-priority secure coding issues as per SEI CERT Secure Coding Rules.

To complete this part of the CREW spreadsheet, an analyst can use an array of SA tools to search for violations of the CERT Secure Coding Rules. An aggregation tool can then be used to combine all SA output together. In the CERT Summary tab, fill out the number of diagnostics found for each of the listed CERT rules. The CERT Summary tab will autofill all the questions in the Security CERT tab. Note that the CERT secure coding violations identified by SA tools are not adjudicated by analysts. Instead, these violations are simply used to estimate the overall risk of the codebase based on their frequency and severity. If CERT secure coding standard violation adjudication is required, a more detailed security analysis can be provided as part of a SCALE analysis [SEI 2023]. The SCALE analysis process is described in more detail in the following *CERT Summary* chapter. The Security CERT tab is shown in Figure 13.1.1.

Figure 13.1.1 The Security CERT Tab

Focus	Risk	W	ID	I	Measurement / Question	Λ	Finding	D	Note
DENSITY C/C++	1.9	0.900	SECTDC01	0	Source lines of code (PSLOC) divided by 1000		382.40	1	
			SECTDC02	1	Diagnostics / KSLOC Priority 1		5.09	0	
			SECTDC03	1	Diagnostics / KSLOC Priority 2		0.26	0	
			SECTDC04	1	Diagnostics / KSLOC Priority 3		2.82	0	
			SECTDC05	1	Diagnostics / KSLOC Priority 4		1.38	0	
			SECTDC06	1	Diagnostics / KSLOC Priority 6		1.89	0	
			SECTDC07	1	Diagnostics / KSLOC Priority 9		6.64	0	
DENSITY C++			SECTDP01	1	Diagnostics / KSLOC Priority 1	C	1.10	0	
			SECTDP02	1	Diagnostics / KSLOC Priority 2	C	4.52	0	
			SECTDP03	1	Diagnostics / KSLOC Priority 3	C	0.10	0	
			SECTDP04	1	Diagnostics / KSLOC Priority 4	C	2.88	0	
			SECTDP05	1	Diagnostics / KSLOC Priority 6	C	3.92	0	
			SECTDP06	1	Diagnostics / KSLOC Priority 9	C	1.27	0	
PRIORITY C/C++	3.9	0.100	SECTPC01	1	Number of Priority 6 diagnostics		724	0	
			SECTPC02	1	Number of Priority 9 diagnostics		2540	0	
PRIORITY C++			SECTPP01	1	Number of Priority 6 diagnostics	C	1500	0	
			SECTPP02	1	Number of Priority 9 diagnostics	C	485	0	
RISK	2.07								

Below is a list of the CERT C and C++ Rules analyzed within CREW

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
PRE30-C	Do not create a universal character name through concatenation	1	Low	Unlikely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
PRE31-C	Avoid side effects in arguments to unsafe macros	1	Low	Unlikely
PRE32-C	Do not use preprocessor directives in invocations of function-like macros	1	Low	Unlikely
DCL30-C	Declare objects with appropriate storage durations	6	High	Probable
DCL31-C	Declare identifiers before using them	1	Low	Unlikely
DCL36-C	Do not declare an identifier with conflicting linkage classifications	4	Medium	Probable
DCL37-C	Do not declare or define a reserved identifier	1	Low	Unlikely
DCL38-C	Use the correct syntax when declaring a flexible array member	1	Low	Unlikely
DCL39-C	Avoid information leakage when passing a structure across a trust boundary	1	Low	Unlikely
DCL40-C	Do not create incompatible declarations of the same function or object	1	Low	Unlikely
DCL41-C	Do not declare variables inside a switch statement before the first case label	2	Medium	Unlikely
EXP30-C	Do not depend on the order of evaluation for side effects	4	Medium	Probable
EXP32-C	Do not access a volatile object through a nonvolatile reference	3	Low	Likely
EXP33-C	Do not read uninitialized memory	6	High	Probable
EXP34-C	Do not dereference null pointers	9	High	Likely
EXP35-C	Do not modify objects with temporary lifetime	2	Low	Probable
EXP36-C	Do not cast pointers into more strictly aligned pointer types	2	Low	Probable

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
EXP37-C	Call functions with the correct number and type of arguments	4	Medium	Probable
EXP39-C	Do not access a variable through a pointer of an incompatible type	2	Medium	Unlikely
EXP40-C	Do not modify constant objects	1	Low	Unlikely
EXP42-C	Do not compare padding data	4	Medium	Probable
EXP43-C	Avoid undefined behavior when using restrict-qualified pointers	4	Medium	Probable
EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic	1	Low	Unlikely
EXP45-C	Do not perform assignments in selection statements	3	Low	Likely
EXP46-C	Do not use a bitwise operator with a Boolean-like operand	3	Low	Likely
EXP47-C	Do not call va_arg with an argument of the incorrect type	2	Medium	Unlikely
INT30-C	Ensure that unsigned integer operations do not wrap	9	High	Likely
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	6	High	Probable
INT32-C	Ensure that operations on signed integers do not result in overflow	9	High	Likely
INT33-C	Ensure that division and remainder operations do not result in divide-by-zero errors	3	Low	Likely
INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of	1	Low	Unlikely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
	bits that exist in the operand			
INT35-C	Use correct integer precisions	1	Low	Unlikely
INT36-C	Converting a pointer to integer or integer to pointer	2	Low	Probable
FLP30-C	Do not use floating-point variables as loop counters	2	Low	Probable
FLP32-C	Prevent or detect domain and range errors in math functions	4	Medium	Probable
FLP34-C	Ensure that floating-point conversions are within range of the new type	1	Low	Unlikely
FLP36-C	Preserve precision when converting integral values to floating-point type	1	Low	Unlikely
FLP37-C	Do not use object representations to compare floating-point values	1	Low	Unlikely
ARR30-C	Do not form or use out-of-bounds pointers or array subscripts	9	High	Likely
ARR32-C	Ensure size arguments for variable length arrays are in a valid range	6	High	Probable
ARR36-C	Do not subtract or compare two pointers that do not refer to the same array	4	Medium	Probable
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object	4	Medium	Probable
ARR38-C	Guarantee that library functions do not form invalid pointers	9	High	Likely
ARR39-C	Do not add or subtract a scaled integer to a pointer	6	High	Probable
STR30-C	Do not attempt to modify string literals	3	Low	Likely
STR31-C	Guarantee that storage for strings has sufficient	9	High	Likely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
	space for character data and the null terminator			
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string	6	High	Probable
STR34-C	Cast characters to unsigned char before converting to larger integer sizes	4	Medium	Probable
STR37-C	Arguments to character-handling functions must be representable as an unsigned char	1	Low	Unlikely
STR38-C	Do not confuse narrow and wide character strings and functions	9	High	Likely
MEM30-C	Do not access freed memory	9	High	Likely
MEM31-C	Free dynamically allocated memory when no longer needed	4	Medium	Probable
MEM33-C	Allocate and copy structures containing a flexible array member dynamically	1	Low	Unlikely
MEM34-C	Only free memory allocated dynamically	9	High	Likely
MEM35-C	Allocate sufficient memory for an object	6	High	Probable
MEM36-C	Do not modify the alignment of objects by calling realloc()	2	Low	Probable
FIO30-C	Exclude user input from format strings	9	High	Likely
FIO32-C	Do not perform operations on devices that are only appropriate for files	2	Medium	Unlikely
FIO34-C	Distinguish between characters read from a file and EOF or WEOF	6	High	Probable
FIO37-C	Do not assume that fgets() or fgetws() returns	6	High	Probable

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
	a nonempty string when successful			
FIO38-C	Do not copy a FILE object	2	Low	Probable
FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call	3	Low	Likely
FIO40-C	Reset strings on fgets() or fgetws() failure	2	Low	Probable
FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	1	Low	Unlikely
FIO42-C	Close files when they are no longer needed	2	Medium	Unlikely
FIO44-C	Only use values for fsetpos() that are returned from fgetpos()	2	Medium	Unlikely
FIO45-C	Avoid TOCTOU race conditions while accessing files	6	High	Probable
FIO46-C	Do not access a closed file	2	Medium	Unlikely
FIO47-C	Use valid format strings	3	High	Unlikely
ENV30-C	Do not modify the object referenced by the return value of certain functions	2	Low	Probable
ENV31-C	Do not rely on an environment pointer following an operation that may invalidate it	2	Low	Probable
ENV32-C	All exit handlers must return normally	6	Medium	Likely
ENV33-C	Do not call system()	6	High	Probable
ENV34-C	Do not store pointers returned by certain functions	2	Low	Probable
SIG30-C	Call only asynchronous-safe functions within signal handlers	9	High	Likely
SIG31-C	Do not access shared objects in signal handlers	9	High	Likely
SIG34-C	Do not call signal() from within interruptible signal handlers	1	Low	Unlikely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
SIG35-C	Do not return from a computational exception signal handler	1	Low	Unlikely
ERR30-C	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure	4	Medium	Probable
ERR32-C	Do not rely on indeterminate values of errno	1	Low	Unlikely
ERR33-C	Detect and handle standard library errors	9	High	Likely
ERR34-C	Detect errors when converting a string to a number	2	Medium	Unlikely
CON30-C	Clean up thread-specific storage	2	Medium	Unlikely
CON31-C	Do not destroy a mutex while it is locked	4	Medium	Probable
CON32-C	Prevent data races when accessing bit-fields from multiple threads	4	Medium	Probable
CON33-C	Avoid race conditions when using library functions	4	Medium	Probable
CON34-C	Declare objects shared between threads with appropriate storage durations	4	Medium	Probable
CON35-C	Avoid deadlock by locking in a predefined order	2	Low	Probable
CON36-C	Wrap functions that can spuriously wake up in a loop	1	Low	Unlikely
CON37-C	Do not call signal() in a multithreaded program	2	Low	Probable
CON38-C	Preserve thread safety and liveness when using condition variables	1	Low	Unlikely
CON39-C	Do not join or detach a thread that was previously joined or detached	3	Low	Likely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
CON40-C	Do not refer to an atomic variable twice in an expression	4	Medium	Probable
CON41-C	Wrap functions that can fail spuriously in a loop	1	Low	Unlikely
CON43-C	Do not allow data races in multithreaded code	4	Medium	Probable
MSC30-C	Do not use the rand() function for generating pseudorandom numbers	2	Medium	Unlikely
MSC32-C	Properly seed pseudorandom number generators	6	Medium	Likely
MSC33-C	Do not pass invalid data to the asctime() function	9	High	Likely
MSC37-C	Ensure that control never reaches the end of a non-void function	3	High	Unlikely
MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	1	Low	Unlikely
MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value	1	Low	Unlikely
MSC40-C	Do not violate constraints	1	Low	Unlikely
MSC41-C	Never hard code sensitive information	6	High	Probable
DCL50-CPP	Do not define a C-style variadic function	6	High	Probable
DCL51-CPP	Do not declare or define a reserved identifier	1	Low	Unlikely
DCL52-CPP	Never qualify a reference type with const or volatile	1	Low	Unlikely
DCL53-CPP	Do not write syntactically ambiguous declarations	1	Low	Unlikely
DCL54-CPP	Overload allocation and deallocation functions as a pair in the same scope	2	Low	Probable
DCL55-CPP	Avoid information leakage when passing a class object across a trust boundary	1	Low	Unlikely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
DCL56-CPP	Avoid cycles during initialization of static objects	1	Low	Unlikely
DCL57-CPP	Do not let exceptions escape from destructors or deallocation functions	3	Low	Likely
DCL58-CPP	Do not modify the standard namespaces	3	High	Unlikely
DCL59-CPP	Do not define an unnamed namespace in a header file	2	Medium	Unlikely
DCL60-CPP	Obey the one-definition rule	3	High	Unlikely
EXP50-CPP	Do not depend on the order of evaluation for side effects	4	Medium	Probable
EXP51-CPP	Do not delete an array through a pointer of the incorrect type	1	Low	Unlikely
EXP52-CPP	Do not rely on side effects in unevaluated operands	1	Low	Unlikely
EXP53-CPP	Do not read uninitialized memory	6	High	Probable
EXP54-CPP	Do not access an object outside of its lifetime	6	High	Probable
EXP55-CPP	Do not access a cv-qualified object through a cv-unqualified type	4	Medium	Probable
EXP56-CPP	Do not call a function with a mismatched language linkage	1	Low	Unlikely
EXP57-CPP	Do not cast or delete pointers to incomplete classes	2	Medium	Unlikely
EXP58-CPP	Pass an object of the correct type to va_start	2	Medium	Unlikely
EXP59-CPP	Use offsetof() on valid types and members	2	Medium	Unlikely
EXP60-CPP	Do not pass a nonstandard-layout type object across execution boundaries	6	High	Probable

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
EXP61-CPP	A lambda object must not outlive any of its reference captured objects	6	High	Probable
EXP62-CPP	Do not access the bits of an object representation that are not part of the object's value representation	6	High	Probable
EXP63-CPP	Do not rely on the value of a moved-from object	4	Medium	Probable
INT50-CPP	Do not cast to an out-of-range enumeration value	2	Medium	Unlikely
CTR50-CPP	Guarantee that container indices and iterators are within the valid range	9	High	Likely
CTR51-CPP	Use valid references, pointers, and iterators to reference elements of a container	6	High	Probable
CTR52-CPP	Guarantee that library functions do not overflow	9	High	Likely
CTR53-CPP	Use valid iterator ranges	6	High	Probable
CTR54-CPP	Do not subtract iterators that do not refer to the same container	4	Medium	Probable
CTR55-CPP	Do not use an additive operator on an iterator if the result would overflow	9	High	Likely
CTR56-CPP	Do not use pointer arithmetic on polymorphic objects	9	High	Likely
CTR57-CPP	Provide a valid ordering predicate	2	Low	Probable
CTR58-CPP	Predicate function objects should not be mutable	3	Low	Likely
STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator	9	High	Likely
STR51-CPP	Do not attempt to create a <code>std::string</code> from a null pointer	9	High	Likely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
STR52-CPP	Use valid references, pointers, and iterators to reference elements of a <code>basic_string</code>	6	High	Probable
STR53-CPP	Range check element access	3	High	Unlikely
MEM50-CPP	Do not access freed memory	9	High	Likely
MEM51-CPP	Properly deallocate dynamically allocated resources	9	High	Likely
MEM52-CPP	Detect and handle memory allocation errors	9	High	Likely
MEM53-CPP	Explicitly construct and destruct objects when manually managing object lifetime	9	High	Likely
MEM54-CPP	Provide placement new with properly aligned pointers to sufficient storage capacity	9	High	Likely
MEM55-CPP	Honor replacement dynamic storage management requirements	9	High	Likely
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer	9	High	Likely
MEM57-CPP	Avoid using default operator new for over-aligned types	2	Medium	Unlikely
FIO50-CPP	Do not alternately input and output from a file stream without an intervening positioning call	3	Low	Likely
FIO51-CPP	Close files when they are no longer needed	2	Medium	Unlikely
ERR50-CPP	Do not abruptly terminate the program	2	Low	Probable
ERR51-CPP	Handle all exceptions	2	Low	Probable
ERR52-CPP	Do not use <code>setjmp()</code> or <code>longjmp()</code>	2	Low	Probable

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
ERR53-CPP	Do not reference base classes or class data members in a constructor or destructor function-try-block handler	1	Low	Unlikely
ERR54-CPP	Catch handlers should order their parameter types from most derived to least derived	6	Medium	Likely
ERR55-CPP	Honor exception specifications	3	Low	Likely
ERR56-CPP	Guarantee exception safety	9	High	Likely
ERR57-CPP	Do not leak resources when handling exceptions	2	Low	Probable
ERR58-CPP	Handle all exceptions thrown before main() begins executing	3	Low	Likely
ERR59-CPP	Do not throw an exception across execution boundaries	6	High	Probable
ERR60-CPP	Exception objects must be nothrow copy constructible	2	Low	Probable
ERR61-CPP	Catch exceptions by lvalue reference	1	Low	Unlikely
ERR62-CPP	Detect errors when converting a string to a number	2	Medium	Unlikely
OOP50-CPP	Do not invoke virtual functions from constructors or destructors	1	Low	Unlikely
OOP51-CPP	Do not slice derived objects	2	Low	Probable
OOP52-CPP	Do not delete a polymorphic object without a virtual destructor	3	Low	Likely
OOP53-CPP	Write constructor member initializers in the canonical order	2	Medium	Unlikely
OOP54-CPP	Gracefully handle self-copy assignment	2	Low	Probable

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
OOP55-CPP	Do not use pointer-to-member operators to access nonexistent members	6	High	Probable
OOP56-CPP	Honor replacement handler requirements	2	Low	Probable
OOP57-CPP	Prefer special member functions and overloaded operators to C Standard Library functions	6	High	Probable
OOP58-CPP	Copy operations must not mutate the source object	3	Low	Likely
CON50-CPP	Do not destroy a mutex while it is locked	4	Medium	Probable
CON51-CPP	Ensure actively held locks are released on exceptional conditions	2	Low	Probable
CON52-CPP	Prevent data races when accessing bit-fields from multiple threads	4	Medium	Probable
CON53-CPP	Avoid deadlock by locking in a predefined order	2	Low	Probable
CON54-CPP	Wrap functions that can spuriously wake up in a loop	1	Low	Unlikely
CON55-CPP	Preserve thread safety and liveness when using condition variables	1	Low	Unlikely
CON56-CPP	Do not speculatively lock a non-recursive mutex that is already owned by the calling thread	1	Low	Unlikely
MSC50-CPP	Do not use <code>std::rand()</code> for generating pseudorandom numbers	2	Medium	Unlikely
MSC51-CPP	Ensure your random number generator is properly seeded	6	Medium	Likely
MSC52-CPP	Value-returning functions must return a value from all exit paths	4	Medium	Probable
MSC53-CPP	Do not return from a function declared <code>[[noreturn]]</code>	2	Medium	Unlikely

Rule	Description	Overall Priority	Severity of Exploitation	Chance of Exploitation
MSC54-CPP	A signal handler must be a plain old function	6	High	Probable

14 CERT Summary

14.1 Overview

In the CERT Summary tab, the analyst enters the number of SA diagnostics for each of the listed CERT rules. The SEI uses a process known as SCALE to complete this part of the CREW analysis. It is well known that every SA tool has unique strengths and weaknesses, and multiple SA tools should be utilized in tandem to gain the most comprehensive perspective on the security risks present in a codebase. Because each analyzer provides its own interface for managing its alerts, it can be challenging to seamlessly use multiple SA tools together during analysis. To address these issues, the SCALE auditing framework aggregates output from commercial, open source, and experimental analysis tools, and maps alerts about possible code flaws from code analysis tools to taxonomies of code flaws. For more information on SCALE and the related tools used during a SCALE analysis, there are several resources available on the SEI's SCALE collection webpage [SEI 2023]. The CERT Summary tab is shown in Figure 14.1.1.

Figure 14.1.1 The CERT Summary Tab

Category	Rule ID	Rule Description	Diagnostic Count	Diagnostic Density
Overview	Total Diagnostics		12182	81.213
	Priority 1 Diagnostics		2367	15.780
	Priority 2 Diagnostics		1825	12.167
	Priority 3 Diagnostics		1114	7.427
	Priority 4 Diagnostics		1627	10.847
	Priority 6 Diagnostics		2224	14.827
	Priority 9 Diagnostics		3025	20.167
Preprocessor C	PRE30-C	Do not create a universal character name through concatenation	0	0.000
	PRE31-C	Avoid side effects in arguments to unsafe macros	0	0.000
Declarations & Initialization C	PRE32-C	Do not use preprocessor directives in invocations of function-like macros	100	0.667
	DCL30-C	Declare objects with appropriate storage durations	20	0.133
	DCL31-C	Declare identifiers before using them	60	0.400
	DCL36-C	Do not declare an identifier with conflicting linkage classifications	80	0.533
	DCL37-C	Do not declare or define a reserved identifier	0	0.000
	DCL38-C	Use the correct syntax when declaring a flexible array member	0	0.000
	DCL39-C	Avoid information leakage when passing a structure across a trust boundary	0	0.000
	DCL40-C	Do not create incompatible declarations of the same function or object	5	0.033
	DCL41-C	Do not declare variables inside a switch statement before the first case label	0	0.000
	EXP30-C	Do not depend on the order of evaluation for side effects	0	0.000
Expressions C	EXP32-C	Do not access a volatile object through a nonvolatile reference	0	0.000
	EXP33-C	Do not read uninitialized memory	20	0.133
	EXP34-C	Do not dereference null pointers	0	0.000
	EXP35-C	Do not modify objects with temporary lifetime	0	0.000
	EXP36-C	Do not cast pointers into more strictly aligned pointer types	17	0.113
	EXP37-C	Call functions with the correct number and type of arguments	0	0.000

A brief overview of the SCALE audit process is provided below.

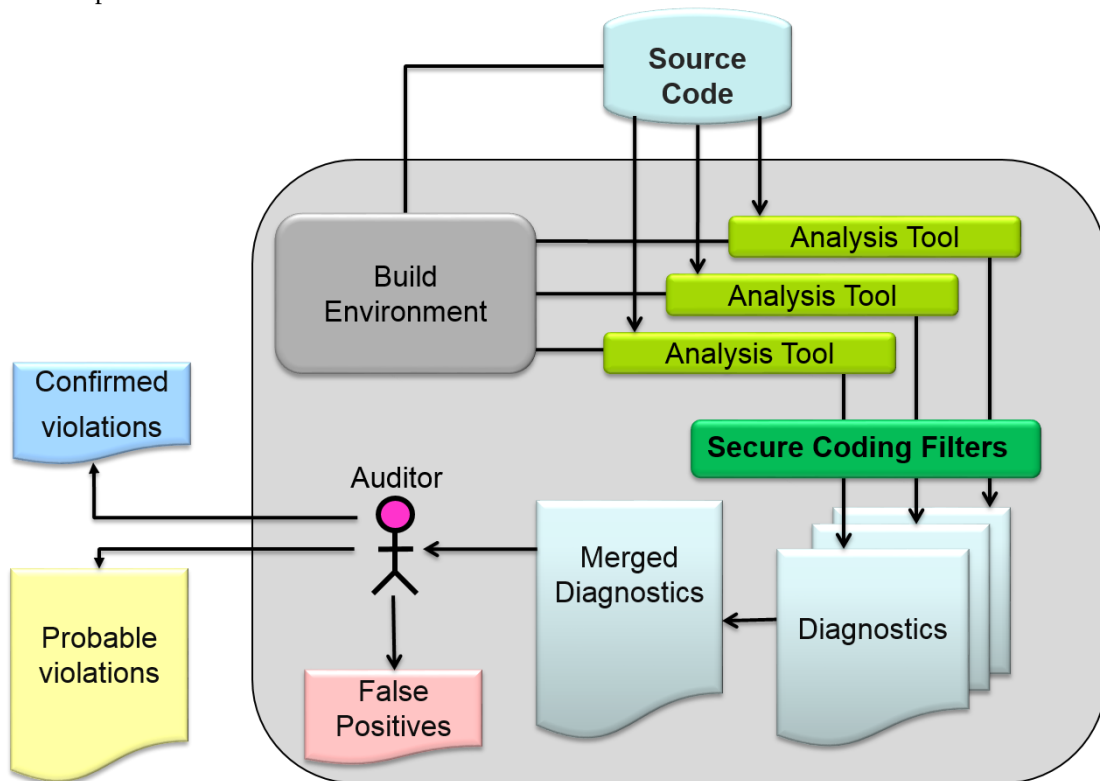
14.2 SCALE Introduction

SCALE (Source Code Analysis Lab) was created in 2010 as a project in the SEI's Secure Coding Initiative (SCI). It has been used to provide high-quality security audits to codebases in C/C++, Java, and Perl.

SCALE includes an application and processes for detecting vulnerabilities in source code. By using output from multiple flaw-finding static code analysis tools, SCALE finds more code defects than any single flaw-finding static code analysis tool would find. However, using multiple tools increases the human work required to handle the large volume of diagnostic output, also known as alert output. Alerts produced by the flaw-finding static code analysis tools must be examined by a human expert who determines whether each alert represents an actual code defect (true positive) or not (false positive). The SCALE system is designed to make this process easier and to automate

much of it. SCALE can also be useful as part of a code development system or simply for code auditing.

SCALE uses filters to screen for alerts that can be mapped to CWEs or CERT Coding rules (i.e., for only alerts that indicate software security issues), thereby reducing the number of alerts that might be examined by a human. SCALE also facilitates the auditor's process of examining alerts, identifying true positives, and saving that information to a database. SCALE provides many scripts and other software to help run static analysis tools, to create a code audit report, and to manage information about the analysis (and about the source code being analyzed). The following image outlines the flow of data in the SCALE process:



In the SCALE process, a source codebase is analyzed using multiple static analysis tools. Static analysis tools analyze the code without executing it, and SCALE uses output from two kinds of static analysis tools: flaw-finding tools and code metrics tools. These tools can be open source or proprietary. Each flaw-finding tool produces output (often copious) containing alerts—that is, problems in the source code identified by the tool. Some, but not all, of these alerts indicate vulnerabilities (such as buffer overflows).

Flaw-finding analysis tools may report problems related to style (such as insufficient comments) or performance (such as call of a slow function when a faster one would suffice) but not related to security. Only alerts related to security are mapped to CERT coding rules. Any alert that doesn't have at least one mapping to either a CWE or CERT coding rule is filtered from the results.

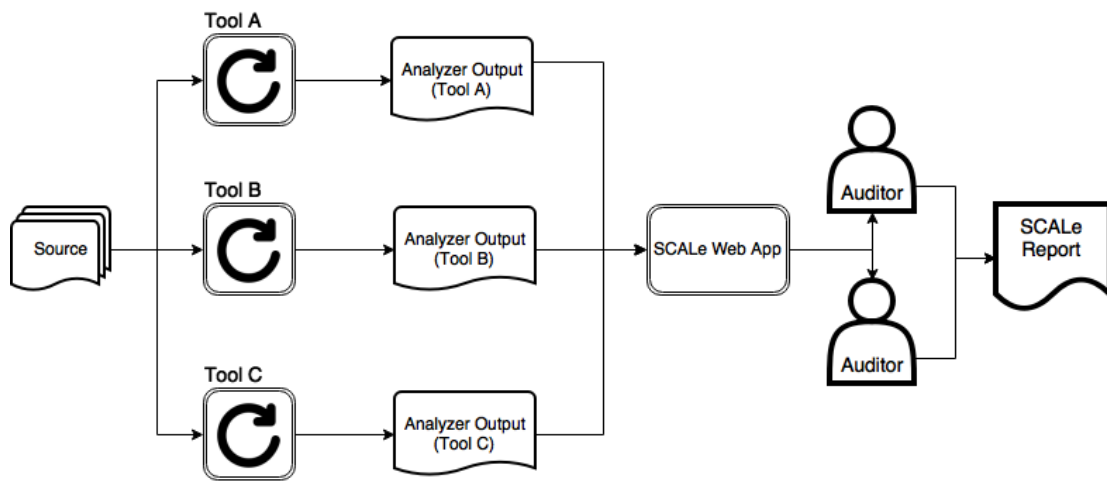
Those alerts and the code metrics are then placed into a database where the alerts are examined by one or more auditors. Each auditor, when examining an alert, must decide if it is correct (a true violation of a secure coding rule), incorrect (a false positive), or if a different verdict applies. Typically, the alerts are too numerous for the auditors to examine all of them. In SCALE, this issue is addressed in multiple ways which can be summarized as follows: The first method prioritizes alerts for manual examination, and auditor effort/time is used auditing the prioritized list in order. A second method partitions alerts into *buckets*; each bucket contains all the alerts associated with a single secure coding rule. For each bucket, the auditor analyzes the alerts until a confirmed violation is found or until the bucket is exhausted. If the auditor finds a confirmed violation, all remaining unconfirmed alerts are marked as suspicious (probable violations), and the auditor moves on to the next bucket. The result of the second method is a classification of the security alerts into true violations, probable violations, and false positives. A third method involves using an automated classifier to partition alerts into expected-true (e-TP), expected-false (e-FP), and indeterminate (I). The automated classifier can be used in conjunction with prioritization of manual examination of indeterminate alerts. The current version of SCALE includes many features for integrating use of classification, but it does not yet perform automated classification.

The SCALE application simplifies the process of auditing alerts. It takes as input the source code and the feedback from the various tools that can be run on the code. With this information, it provides a web-based interface to the alerts, allowing auditors to quickly view all information associated with the alert, including the source code it applies to. It makes auditor work more efficient by fusing alerts into a single meta-alert view, where alerts for the same file path, line, and condition (that is, CERT rule or CWE) are shown together. It also enables auditors to check if previous audit determinations or notes have been provided for the same checker, file path, and line but different condition(s). SCALE allows auditors to mark a meta-alert's (single) verdict, any combination of supplemental verdict(s), add notes, and add a flag. SCALE enables auditors to export the alerts (and meta-alerts, code metrics, and all other SCALE database data) for outside work, such as for classifier development or for building a spreadsheet of the alerts marked as true positives.

SCALE enables users to create alert prioritization schemes to order alerts in the GUI according to user-editable mathematical formulas, using meta-alert-related fields such as classifier-derived confidence, risk, and cost. (Confidence that a meta-alert is true is not the only important factor. Other important factors include how severe of a security risk a code flaw condition poses, and an estimate how much it costs to fix the flaw. A priority scheme formula might use each of these and additional metrics.)

Advanced users can upload files to provide new data that can be used in these prioritization formulas. SCALE filters can be used to limit the view to alerts of interest (e.g., a particular taxonomy, condition, or file path). SCALE maintains a history of audit determinations.

Finally, SCALE code includes a script that creates a "sanitized" version of an exported SCALE database, anonymizing fields (and deleting some fields) that may possibly contain sensitive data, while retaining sufficient data for machine learning classification features. (However, the sanitization functionality is not completely updated for the latest version of the exported database file exports, so for now, users should not depend on the sanitizer to secure their sensitive data.)



The diagram above gives a high-level overview of the SCALe audit process. A source codebase is analyzed with various static analysis tools. Output from the tools is uploaded to the SCALe web app. Auditors use the web app to evaluate the alerts, determining if alerts are true violations of conditions (such as CERT Secure Coding rules or CWEs). Auditors can make other determinations than simply true or false, as SCALe now provides auditors all the options for labeling primary and supplemental determinations (and notes) as described in the paper [Static Analysis Alert Audits: Lexicon & Rules](#). The results of auditor evaluations can be exported into a database or other file, and that can be compiled into a report. The SCALe web app recognizes outputs from many popular static analyzers.

14.3 SCALe User Roles

The SCALe process involves developers, analysts, auditors, and administrators. The number of people performing each role is variable. For example, the SCALe process can be done by a single person performing all roles or by a large organization with teams of people for each role.

Developers are responsible for fixing any vulnerabilities found in the software, and they serve as the foremost authorities on the software's implementation. They also are responsible for providing the source code (to be analyzed) to the SCALe app.

Analysts are responsible for running appropriate static analysis tools on the software. They may rely on the developers for wisdom about properly building the software. Analysts are responsible for providing tool output to the SCALe app.

Auditors evaluate diagnostic alerts, deciding which are confirmed violations and which are false positives. In this role, they are the primary users of the SCALe app. To correctly evaluate alerts, auditors must be familiar with the SEI CERT Coding Standards. These are available on the web, and the [C](#) and [Java](#) standards have been published by Addison-Wesley. CERT also offers courses in [C and C++](#) and Java for secure coding.

Administrators are responsible for installing and maintaining the SCALE app as well as any tools required by the analysts. The host running the SCALE app need not be directly accessed by anyone except the administrator.

14.4 SCALE Audit Basic Instructions

These instructions assume the source codebase is suitable for SCALE. A suitable codebase must be analyzable by the tools supported by SCALE. These tools require that the codebase must be compiled. Code that is missing internal libraries or headers cannot be analyzed in SCALE until these missing components are supplied.

While compilation is a requirement, the codebase does not need to link properly or run properly. Many static analyzers, including Coverity and Fortify, require access to a functioning build of the source. These analyzers rely on compiler processes invoked during the build (for example, GCC) to parse the source. Some tools process the source directly, but often with lower quality results.

The basic steps for analyzing a codebase through SCALE are as follows:

1. Perform a Test Build.
2. Run the Static Analysis Tools associated with your source codebase's platform.
3. Upload your codebase and analysis tool output to the SCALE web app.
4. If the codebase has undergone a previous SCALE audit (perhaps as an earlier version), you will want to apply its previous determinations.
5. Validate the SCALE Diagnostics.
6. Construct the SCALE Audit Report.

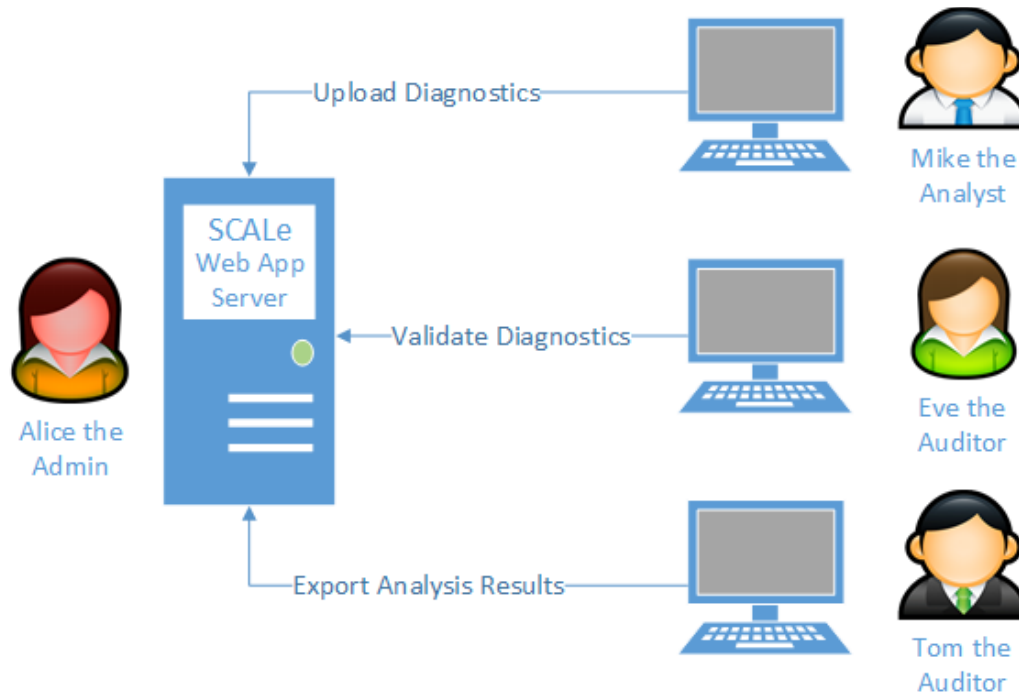
Note, the SEI has [more information](#) about SCALE, including links to online instructional SCALE videos (outdated for the current version), details about auditing (mostly outdated), and information about SCALE services, the SCALE seal, and the SCALE certificate.

14.5 The SCALE Web App

The SCALE web app facilitates the auditing of alerts from various static analyzers. It is a Ruby on Rails application, accessed through a web browser. The main use cases for this application are as follows:

- Uploading alerts from static analyzers
- Inspecting alerts for violations of CERT Secure Coding rules or instances of Common Weakness Enumeration weaknesses (CWEs)
- Exporting analysis results

This diagram illustrates how users interact with the web app.



Here is a table overview of the actors in this diagram with their responsibilities:

Actor	Role
Alice the Admin	Installs the web app on a dedicated server and configures access for the other users
Mike the Analyst	Runs static analysis tools on source code and uploads the results to the web app
Eve the Auditor	Inspects the results uploaded by Mike, using the web app, for true violations of conditions (e.g., CERT Secure Coding rules) or instances of conditions (e.g., CWEs)
Tom the Auditor	Assists Eve and, for reporting purposes, exports the alerts from the web app after auditing is complete

14.5.1 Uploading alerts from Static Analyzers

This section documents how users interact with the web app. These interactions cover the responsibilities of Mike, Eve, and Tom in the diagram above.

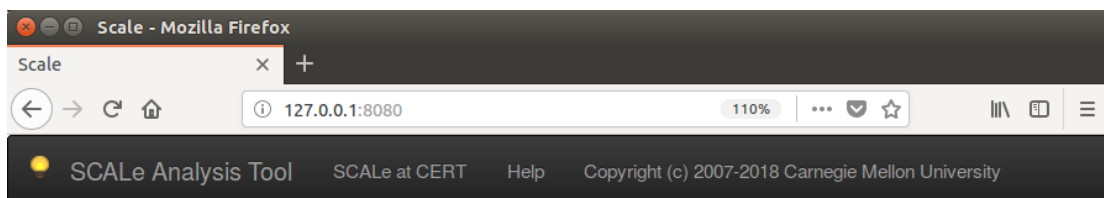
The following browsers have been tested with the SCALE web app:

Browser	Version
Firefox	40.0.3
Internet Explorer	11.0
Chrome	44

Mike the Analyst can use the web app to upload the output from static analysis tools. Mike will need the following before doing so:

- The outputs from all desired static analysis tools
- The source code to be analyzed in a zip archive
- The previous two items on a machine with connectivity to the SCALE web app
- The network address, port, username, and password for the web app from Alice

Mike connects to the web app using the details supplied by Alice. He arrives at the homepage and sees a list of existing projects maintained by the web app. He then creates a new project for this audit by clicking the **New project** button:



Active SCALE Projects

Name	Description	Project Options
dos2unix		   



[New project](#)

SCALE Copyright (c) 2007-2018 Carnegie Mellon University. All Rights Reserved. See COPYRIGHT file for details.

He then specifies a name and description for the project and clicks the **Create Project** button:

The screenshot shows a web browser window titled "Scale - Mozilla Firefox" with a single tab labeled "Scale". The address bar shows the URL "127.0.0.1:8080/projects/new". The page header contains the text "SCALE Analysis Tool", "SCALE at CERT", "Help", and "Copyright (c) 2007-2018 Carnegie Mellon University". The main content area is titled "New project" and contains three input fields: "Name:" with a placeholder "Project Name", "Description:" with a placeholder "Description of project", and a "Create Project" button. Red arrows point to each of these elements. At the bottom of the form, there is a copyright notice: "SCALE Copyright (c) 2007-2018 Carnegie Mellon University. All Rights Reserved. See COPYRIGHT file for details."

Next, Mike uploads the source code and the outputs from the static analyzers. To upload the source, he clicks the **Browse...** button next to the label **Archive containing src:**. SCALE can import source as a zip file or a tarball (.tar.gz or .tar.bz2). Then he selects the archive containing the source code from his local machine.

SCALE needs to normalize the file name references for the source code with the file name references from the static analysis tool. The section of the manual "Manual Project Creation" has details on that, including how to create parts of a project manually and ways to resolve file paths if the import using the SCALE GUI has problems resolving file paths (to match the source code file paths and static analysis output file paths). Hopefully importing source code and static analysis output via the GUI will work correctly though, as that is simpler.

Next, he uploads the output from each tool using the appropriate row in the table below the source upload dialog. The first column of this table identifies general-purpose static analysis tools by tool name and language in the format **ID / TOOL NAME / LANGUAGE**. The ID uniquely identifies a Tool/Language combination. For example, the row **11 / gcc / c** identifies the "gcc" tool (compiler) for C/C++ source code. The checkbox next to this identified tool and language should be checked, then the output uploaded by clicking the **Browse...** button next to the **Tool output:** label.

Here is an example of what this page might look like when Mike uploads source code and output for two general flaw-finding static analysis tools (gcc and Rosecheckers):

Source

Archive containing src: dos2unix-7.2.2.tgz

Tool & Platform	Diagnostics & Metrics	Script Output
☒ 11 / gcc / c	Tool output: Browse... gcc.txt	
☒ 12 / rosecheckers / c	Tool output: Browse... rosecheckers.txt	
☐ 13 / msvc / c	Tool output: Browse... No file selected.	

To upload output from code metrics static analysis tools (e.g., Lizard, Understand, or CCSM), similarly the tool must be selected, and the user should browse to select the tool output.

☐ 91 / lizard / metric	Tool output: Browse... No file selected.
☒ 92 / ccsm / metric	Tool output: Browse... dos2unix_ccsm.c



Next, Mike clicks the **Create Database** icon at the top right of the page (above the table for selecting output uploads):

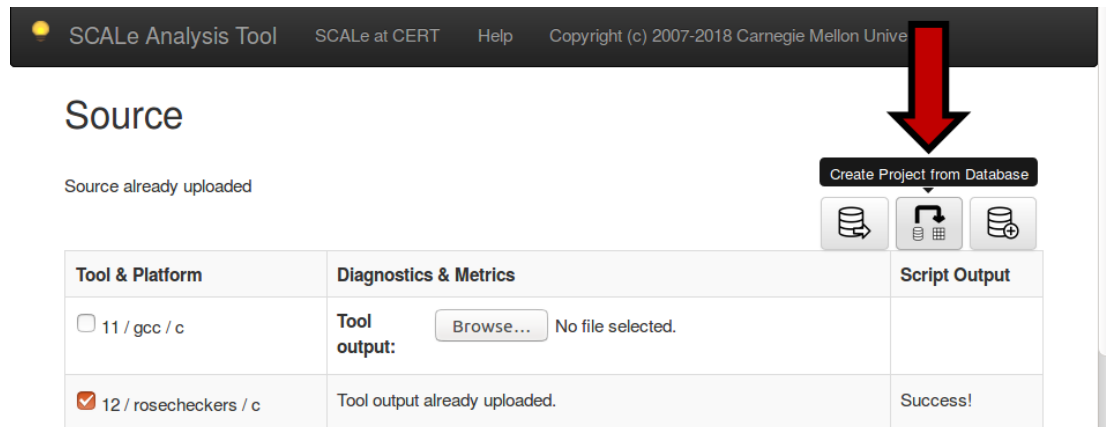
SCALE Analysis Tool SCALE at CERT Help Copyright (c) 2007-2018 Carnegie Mellon University

Source

Archive containing src: dos2unix-7.2.2.tgz

Tool & Platform	Diagnostics & Metrics	Script Output
☐ 11 / gcc / c	Tool output: Browse... No file selected.	
☒ 12 / rosecheckers / c	Tool output: Browse... rosecheckers.txt	

The creation process consolidates the uploaded outputs into a single database. The page refreshes, and any errors that occurred during this processing are displayed in the "Script Output Column" in the row corresponding to the tool output that caused the error. If no errors occur, Mike creates the project next by clicking the **Create project from database** icon at the top of the page (image below shows text that appears when mouse hovers over the icon):



He could also click the icon on the left titled **Download database** (hover mouse over icon to see text) to download a copy of the alert database to his system. After creating the project, Mike's job is done.

14.5.2 Inspecting Alerts for Violations of CERT Secure Coding Rules

After Mike the Analyst uploads the alerts to the web app, it is Eve the Auditor's job to inspect these alerts. To do so, Eve navigates to the SCALe web app and logs in using information obtained from Alice. She arrives at the web app homepage where she clicks on the name of the project created by Mike:

Active SCALE Projects

Name	Description	Project Options
dos2unix		   
dos2unix_v2	second version of codebase	   

New project

14.5.3 The Alert Viewer

Clicking the project name takes Eve to the Alert Viewer page. This page allows her to view all the alerts produced by the static analysis tools. She can filter the alerts using various criteria. Information pertinent to auditing is provided for each alert. For example, the location of the alert in the source code is provided. The source code associated with each alert can also be viewed in the browser. A link is provided to the CERT Secure Coding rule associated with the alert. Eve's job is to determine if each alert is indeed a violation of its corresponding Secure Coding rule.

Here is a screenshot of the alert viewer:

The screenshot displays the SCALE Analysis Tool interface for a project named 'dos2unix'. The interface is divided into several sections:

- Header Menu:** Contains links for 'SCALE Analysis Tool', 'SCALE at CMU', 'Classifiers', 'Prioritization Schemes', and 'Help'. It also shows the copyright information: 'Copyright (c) 2007-2018 Carnegie Mellon University'.
- Alert Filters:** A toolbar with various filters including 'All IDs', 'Line', 'Checker', 'Tool', 'Condition', and 'Taxonomy'. It also includes a 'Filter' button and a 'Fused View' toggle.
- Middle Menu:** Displays 'Showing 1 to 10 of 253 (Diagnostic per page: 10)' and a 'Set selected to:' dropdown. It also shows the 'Classifier Selected: None Selected' and a 'Classify' button.
- Alert List:** A table listing alerts with columns for ID, Flag, Verdict, Supplemental, Notes, Previous, Path, Line, Message, Checker, Tool, Condition, Title, Confidence, Alert Pri, Sev, Lk, Rem, Pri, Lev, CWE, Lk. The table shows several alerts related to CWE-359 and CWE-679.
- Source Code Viewer:** Displays the source code for the 'MAIN' function, showing a loop that iterates over 'main' and 'target' arrays.

From top to bottom:

- The black **Header Menu** toolbar allows you to edit and export projects ("SCALE Analysis Tool"), to create/edit/select Classifier schemes, to create/edit/select Prioritization schemes, and provides information about SCALE.
- The light-blue **Alert Filters** toolbar allows you to filter the displayed alerts by various criteria.
- The **Middle Menu** allows users to modify the alert list size, displays the Classifier scheme name if one has been selected, and provides a button for the user to manually start automated classification of alerts. (Note: this functionality is not completely implemented, and selecting the button does not classify the alerts. It currently simply inserts example confidence values, not real ones.)
- The **Alert List** section displays the meta-alerts, as fused alerts (that show a meta-alert ID on the yellow-colored-background row that audit determinations can be made on) or as single alerts (in a white-background row) that show a 'displays' ID.
- *Detail:* A 'displays' ID is unique for each combination of condition ID and alert ID. Condition ID is unique per flaw type (e.g., one ID for CWE-190 and another ID for INT31-C). Alert ID is unique for a particular warning from a tool for a particular flaw type the tool identifies with its "checker", for a particular line number and file path. Fused alerts can be expanded by selecting editable fields on the meta-alert row, and then the alerts that were fused show on separate rows (with aqua-colored background) below that. If there are many alerts, they appear on multiple pages. To navigate these pages, click the **Next/Previous** links near the top of the viewer or click on the desired page number.

- In 'Fused view', each top-level (not expanded) row of the 'Alert List' shows a meta-alert
 - it shows the alert ID (aka 'diagnostic_ID') if the meta-alert has only one alert, meaning no alert fusion happened)
 - it shows the meta-alert ID if the meta-alert contains fused alerts, and can be expanded to show the single alerts it contains
- In 'Unfused view', each row of the 'Alert List' shows an alertCondition
 - an alertCondition is a single alert with information about only one of the conditions its checker maps to
 - If that alert's checker maps to multiple conditions, there will be a separate row in the 'Alert List' for each of the conditions it maps to.
- The **Source Code Viewer** section at the bottom displays the code associated with a given meta-alert.

The following table describes the fields displayed in the Alert Viewer section (not all of these are visible in the screenshot above).

Alert Property Name	Description	Possible Values
ID	A unique ID associated with this meta-alert (could be multiple alerts fused or a single alert)	Positive integer
Flag	A binary value associated with the alert. You can click on the box in the viewer to toggle the flag. The meaning of the flag is user defined. One possible meaning is to indicate "suspicious" alerts, that have not been manually audited, but warrant attention for various reasons.	Flagged or not flagged
Verdict	The current conclusion (primary verdict) with respect to whether this alert violates the associated Secure Coding	Unknown, Complex, Dependent, False, True (actual value used in data structure is a positive integer)

Alert Property Name	Description	Possible Values
	rule (you can click on this field in the viewer to change it). Verdicts are defined in the IEEE Secure Development conference paper Static Analysis Alert Audits: Lexicon & Rules .	
Supplemental	Supplemental verdicts. Supplemental verdicts are defined in the IEEE Secure Development conference paper Static Analysis Alert Audits: Lexicon & Rules .	Ignored, Dead, Inapplicable Environment, (high, medium, low) Dangerous Construct
Notes	Notes by auditor(s) about the determinations (primary and/or supplemental), alert, meta-alert, checker, condition, or language. Intention of field is to help later auditors reviewing same or similar issues.	Any string value
Previous	The number of changes to the determination fields (flag, verdict, supplemental, notes) for this alert. If non-zero, the number links to a table showing a change log for this alert.	0 or a positive integer
Path	The path of the source file where the violation occurs	A relative directory path
Line	The line number in the source file where the alert occurs; selecting this field causes the snippet of source	Positive integer

Alert Property Name	Description	Possible Values
	code that triggered the alert to appear in the Source Code Viewer section	
Message	The message issued by the static analysis tool	Any string value. If hyperlinked text "More" or "Secondary message(s) set" exists, then a set of secondary messages exists for this message, providing additional clues about potentially problematic data/control/type flow related to the message. Selecting the hyperlink will cause those messages to show in a popup window. If separate rows of fused alerts have the same checker ID and message, their secondary message sets differ.
Checker	The name of the checker in the static analysis tool that triggered the alert. This is hyperlinked, and selecting hyperlink leads to list of all meta-alerts for that line of that file AND that checker (there may be multiple conditions, e.g., a CWE and a CERT rule both mapped to that checker ID).	Any string value, with hyperlink as specified in cell to left.
Tool	The name of the static analysis tool	gcc, rosecheckers, etc.
Condition	The identifier of the condition (e.g., CERT Secure Coding rule or CWE) that this alert potentially violates; se-	See https://securecoding.cert.org

Alert Property Name	Description	Possible Values
	lecting this field causes the condition associated with the alert to appear in a new browser window or tab	
Title	The title of the condition this alert indicates a potential violation of (e.g., for CERT Secure Coding rules) or indicates a potential instance of (e.g., for CWEs)	See https://securecoding.cert.org
Confidence	Placeholder field for confidence that meta-alert is True. Not currently used by SCALE web app.	
Alert Priority (Alert Pri)	Priority of making a manual determination of the verdict for the meta-alert, as determined using a prioritization scheme.	
Severity (Sev)	The severity of the consequences of the CERT Secure Coding rule being ignored	
Likelihood (Lik)	The likelihood that a flaw introduced by ignoring the CERT Secure Coding rule can lead to an exploitable vulnerability	
Remediation (Rem)	The expense required to modify the code to comply with the CERT Secure Coding rule	

Alert Property Name	Description	Possible Values
Priority (Pri)	The priority of the violated CERT Secure Coding rule	
Level (Lev)	The level of the violated CERT Secure Coding rule	
CWE likelihood (CWE_Lik)	The likelihood (as defined by MITRE) of the CWE, in the Common Consequences section. "The Likelihood provides information about how likely the specific consequence is expected to be seen relative to the other consequences in the list. For example, there may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact."	E.g., see https://cwe.mitre.org/data/definitions/190.html (Common Consequences section)

14.5.4 The Auditing Workflow

The meta-alerts in the web app indicate potential violations of CERT Secure Coding rules or instances of CWEs. Eve's job is to determine if these meta-alerts are true positives (actual violation of the rule), false positives (not a violation of the rule), or any other correct primary or supplemental determination. The auditor should examine a meta-alert and determine the appropriate primary verdict, supplemental verdict(s), and add text to the Notes field to provide information as needed for future analysts. When examining a particular meta-alert, look at any audit determinations and/or notes for the same checker(s) at this location, to save time by using previously determined related information. Also, examine all messages, including all sets of secondary messages, to use clues from the tools about the condition in question. There are two methods CERT recommends for auditing:

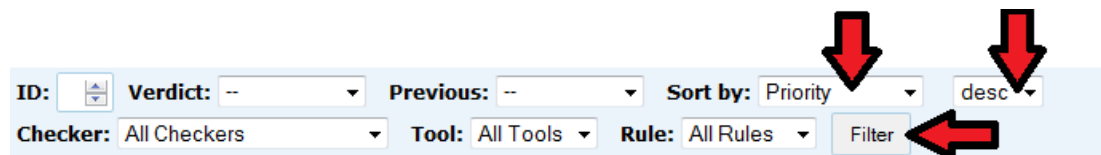
Address meta-alerts according to (filtered and prioritized) priority level. The auditor should be knowledgeable about the condition(s) they audit, and if audited conditions are frequently switched extra time may be required for the auditor to refresh themselves on the condition being audited.

Use bucketing method: One recommended way to use the bucketing method is to structure auditing work by condition. That is, Eve analyzes meta-alerts that correspond to a single condition.

Conditions (e.g., CERT coding rules) with high priority can be inspected first (see "Priority (Pri)" in the table above). She inspects the meta-alerts one by one until she finds a true positive. Any false positives along the way are assigned the False verdict. If a true positive is discovered, she assigns it the True verdict, then flags any remaining uninspected meta-alerts to indicate a Suspicious verdict. After completing this process, Eve moves on to another code condition (CERT Secure Coding Rule or CWE).

Sorting the Alerts by CERT Priority

Note that CERT priority (defined per CERT rule by CERT) is different than alert priority (determined by a prioritization formula). To sort the alerts by CERT Priority, Eve selects CERT Priority in the Sort By drop-down in the Alert Filter toolbar, chooses desc in the adjacent box to indicate a descending sort, and clicks Filter. The resulting alerts will be sorted by priority, beginning with the highest-CERT priority alerts (a larger CERT priority indicates a more severe violation of CERT coding rules).



Filtering by Condition

To filter the meta-alerts by rule, Eve clicks on the Condition drop-down in the Alert Filter toolbar, selects the desired rule, and clicks Filter. For example, the following shows all meta-alerts that potentially violate ERR33-C. Other filters can be applied in a similar fashion.

Auditing an Alert

Eve can use the web app in the following ways to facilitate auditing a given meta-alert:

- Click on the Condition link to review the associated Secure Coding rule or CWE.
- Read the alert message (one or more) in the Message column.
- Click the hyperlinked line number to review the source code associated with the alert in the source code viewer at the bottom of the page.
- If present, click the More link or links in the Message column to review additional information about alert(s) from the analysis tool(s).

The source code viewer uses GNU Global to render the code in HTML. Global also provides simple navigation through the code, for example, to find declarations of variables, functions, and classes. Eve can use this view to explore the source code. Using these resources and her knowledge of the programming language at hand, Eve can decide whether the alert is a true positive or false positive.

While analyzing the potential flaw, auditors can add notes to the Notes field by clicking in the field and then entering text, as shown below. The notes can help future auditors to work with

alerts for this code region and condition, by communicating what has been analyzed and learned so far.

dict	Supplemental	Notes
known]	Edit	Variable X may have integer overflow, must investigate 'else' conditional
known]	Edit	Variable Y appears to be handled safely.
known]	Edit	0
known]	Edit	0

Updating Alert Verdicts

Once Eve concludes regarding a meta-alert (a single alert diagnostic or multiple fused alerts), she sets the Verdict field appropriately. This table describes the possible verdicts:

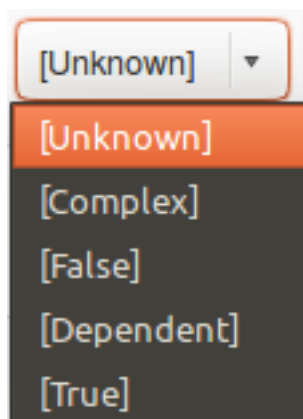
Verdict	Description
Unknown	(Default value before auditing) It is unknown if the meta-alert is a true positive or false positive
Complex	The alert is too difficult to judge in a reasonable amount of time and effort. Each organization should document what is a reasonable amount of time and effort. Preferably, an approximate time spent auditing would be documented with each Complex alert. See Audit Rule 2 in Static Analysis Alert Audits: Lexicon & Rules.
Dependent	The alert indicates a code flaw which could only be true if a different coding flaw that is marked True or “Dangerous construct” occurs earlier

Verdict	Description
	in the code execution. See Audit Rule 3 in Static Analysis Alert Audits: Lexicon & Rules for more information. This determination should include a reference to the related alert, via the alert's unique identifier.
False	The code in question does <i>not</i> violate the condition indicated by the alert.
True	The code in question violates the condition indicated by the alert.
Flagged	Suspicious, using one of the audit methods. The meta-alert has not been manually inspected because another alert violating the same Secure Coding rule was found to be a true positive.

To set the Verdict field, Eve clicks on the text inside the Verdict column associated with the alert. This click creates a drop-down menu.

	ID	Flag	Verdict	Previous	Path
	23	[]	[Unknown] ▼	Unknown	/dos2unix-7.2.2/common.c

A second click will open the drop down, allowing Eve to select the desired verdict. Suppose the alert at hand is a false positive, warranting a **False** verdict.



The alert now has the desired verdict.

☐	ID	Flag	Verdict	Previous	Path
☐	23	[]	[False]	Unknown	/dos2unix-7.2.2/common.c

Multiple alerts can be updated simultaneously. To do so, Eve selects the checkbox by each alert she wants to update, sets values for the Verdict and Flag fields in the drop-down menus next to **Set all selected to** at the top of the page, and clicks **Update**.

In the following screenshot, all selected alerts will have their Verdict field set to **True** after Update is clicked:

Set all selected to: True ▾ -- ▾ Update

☐	ID	Flag	Verdict	Previous	Path	Line
☒	124	[]	[Unknown]	Unknown	/dos2unix-7.2.2/common.c	915
☐	724	[]	[Unknown]	Unknown	/dos2unix-7.2.2/common.c	1876
☒	1224	[]	[Unknown]	Unknown	/dos2unix-7.2.2/dos2unix.c	490
☒	1824	[]	[Unknown]	Unknown	/dos2unix-7.2.2/unix2dos.c	498
☐	124	[]	[Unknown]	Unknown	/dos2unix-7.2.2/common.c	453

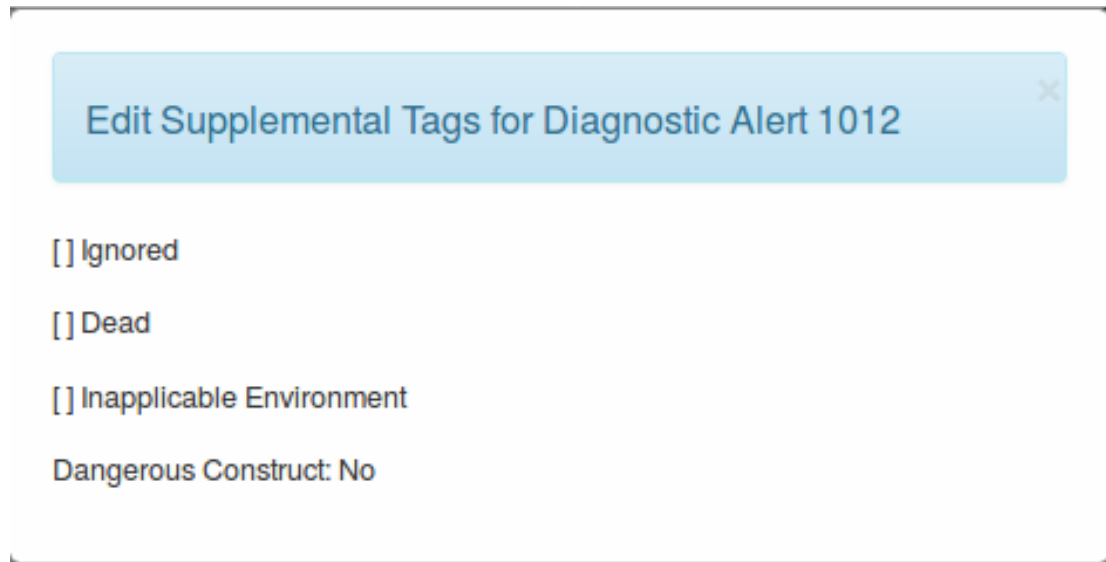
All visible alerts can be selected and deselected by clicking the checkbox in the header row. The number of alerts shown per page can be changed by updating the **Diagnostics per page** drop-down menu and clicking **Go**.

Flagging Alerts

Alerts can also be **flagged**. Eve might flag a diagnostic if she wants to revisit it later. An alert is flagged or unflagged by clicking the box in the **Flag** column. Multiple alerts can be flagged or unflagged, similarly to the way they are simultaneously updated in the Verdict field, using the **Set all selected to** dialog. The second drop-down menu controls the flag status of the selected alerts.

Updating Supplemental Verdicts

Once Eve concludes regarding a diagnostic's supplemental verdict, she sets the Supplemental Verdict field appropriately. A pop-up window appears when she selects the Supplemental Verdict field for a meta-alert, and she can choose one or more supplemental verdicts. Although only one primary verdict can be chosen, any number of supplemental verdicts can be selected. Supplemental verdicts can be set for: Ignored, Dead, Inapplicable Environment, and (high, medium, or low) Dangerous Construct. After she completes making her Supplemental Verdict selections, Eve closes the popup by selecting the "X" at the top-right of the popup window.



Adding Notes

In the Notes field of the meta-alert, Eve adds textual information of use to subsequent auditors (including possibly herself, returning to look at the alert in the future). Her notes may contain her findings relevant to the alert, checker, condition, or language about control flow, data flow, type, alert, checker, condition, or language.

Selecting a Prioritization Scheme

Prioritization scheme selection allows users to prioritize static analysis alerts using factors they care about. The formulas can combine classifier confidence and other values (e.g., risk, cost, etc.) used by the system, using math symbols including: '*', '/', '+', '-', '(', and ')'

To create a new prioritization scheme, select 'Create New Scheme' from the top menu bar's 'Prioritization Schemes' option.

SCALe Analysis Tool SCALe at CERT Classifiers ▾ Prioritization Schemes ▾ Help

Project: dos2unix

[New Diagnostic](#)

priority1
priority2
priority3
Create New Scheme

All IDs + Verdict: --

Next, select each taxonomy tab and enter a formula (or even a single weighted field) to calculate priority for meta-alerts for conditions in that taxonomy. Note that classifier-derived confidence and user-uploaded fields can be used in formulas for all the taxonomies.

Weights (1 or higher) must be selected on the left, for each field that will be used in the formula. Numbers and mathematical operators can also be entered using the keyboard.

The screenshot below shows a warning that appears if the button 'Generate the Formula' is selected before each taxonomy has a formula.

Create New Scheme

Name: myPrioritizationScheme1

Instructions

CWES

CERT_RULES

cwe_likelihood1

confidence2

Formula for CWES

() * + / - cwe_likelihood

(confidence*2)+cwe_likelihood

Please check the tabs to makes sure each taxonomy has a formula

Generate The Formula

Prioritization Formula:

☐ Save Priority

Cancel

Run Priority

Name the prioritization scheme, e.g., in the figure above the name is 'myPrioritizationScheme1'. To save the scheme, select 'Save Priority'.

×

Create New Scheme

Name: myPrioritizationScheme1

Instructions
CWES
CERT_RULES

cert_severity 2

cert_likelihood 1

cert_remediation 1

cert_priority 0

cert_level 0

confidence 2

Formula for CERT_RULES

() * + / - cert_severity

$$(cert_severity*2+cert_remediation)*confidence*2$$

Generate The Formula

Prioritization Formula:

$$IF_CWES((confidence*2)+cwe_likelihood)+IF_CERT_RULES((cert_severity*2+cert_remediation)*confidence*2)$$

✔ **Save Priority** Priority Scheme Saved

Cancel
Run Priority

Run the prioritization by selecting the button 'Run Priority' at the bottom-right of the popup. This will calculate values and put them in the 'Alert pri' ("Alert Priority") field for the meta-alerts, as shown below.

Confidence	Alert Pri	Sev	Lik	Rem
13.69	8	1	1	2
35.42	8	1	1	2
95.93	8	1	1	2
6.3	8	1	1	2
76.49	8	1	1	2
38.38	8	1	1	2
35.84	8	1	1	2
38.08	8	1	1	2
59.08	16	2	1	2
23.43	16	2	1	2

To order meta-alerts by values from this priority scheme, in the filter section for 'Sort by' select 'Alert Priority', and for 'Sort direction' select 'desc' (descending). (See screenshot below with these options selected.) Select the button 'Filter' for the desired prioritization ordering.

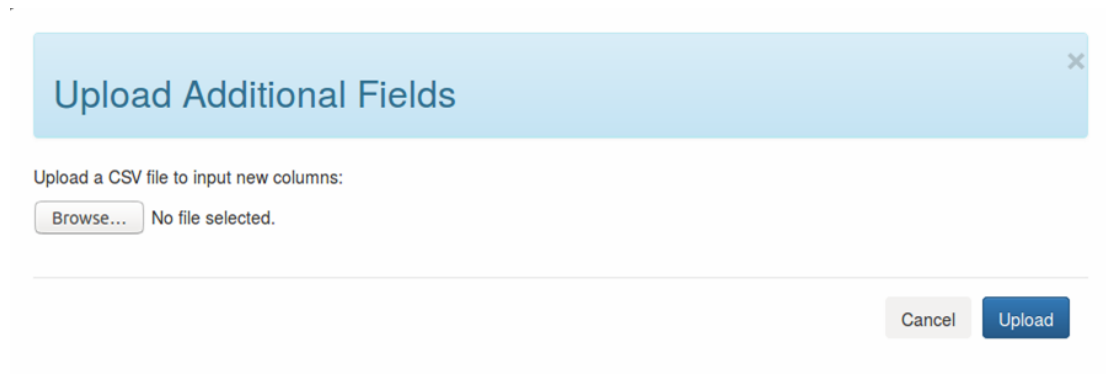
The screenshot shows a filter section with the following settings:

- All IDs: [dropdown]
- Verdict: [dropdown]
- Previous: [dropdown]
- Path: [dropdown]
- Line: [dropdown]
- Checker: All Checkers [dropdown]
- Tool: All Tc [dropdown]
- Condition: All [dropdown]
- Taxonomy: View All [dropdown]
- Sort direction: desc [dropdown]
- Sort by: Alert Priority [dropdown]
- Filter button

After saving and running a new prioritization scheme, on return to the Prioritization menu drop-down, users do not initially see the full formula at the bottom of the popup window for the previously saved scheme. It does show the sub-formulas in each tab, still. Also, if the user selects the button 'Generate The Formula', it will show the full formula.

Uploading additional fields

Users can upload additional fields. Select the header menu item 'Upload New Fields'. The popup window shown below will appear.

The image shows a light blue rectangular popup window titled 'Upload Additional Fields' in a bold, dark blue font. In the top right corner of the title bar is a small 'x' icon for closing the window. Below the title bar, the text 'Upload a CSV file to input new columns:' is displayed. Underneath this text is a file selection interface consisting of a 'Browse...' button and the text 'No file selected.'. At the bottom right of the popup, there are two buttons: a light gray 'Cancel' button and a dark blue 'Upload' button.

Currently, the user-uploaded fields can be used in prioritization formulas but cannot be viewed in the GUI. (Future versions of SCALE will include the fields in the GUI.)

The upload-fields option is currently for advanced users that can work with SQL databases and have or can generate values for new fields, e.g., based on advanced logic using the other alert fields or based on proprietary data. However, an extended and more user-friendly version of the concept could be instantiated without technical difficulty beyond standard development, to enable less-technical users to generate values for new fields using mathematical formulas, advanced logic, and data beyond that provided in the initial SCALE database.

The comma separated value (CSV) uploaded file should have the following:

- one line for every meta-alert ID that exists in the project
- left-most field holds meta-alert ID
- top row holds labels for each field, with left-most top field being equivalent to `meta_alert_id`
- the rest of the rows should hold a value for each field, for each meta-alert

After the new fields have been uploaded, they auto-populate the prioritization scheme interface, below the initially populated fields.

Create New Scheme

Name:

Instructions

CWES

CERT_RULES

cwe_likelihood

confidence

Formula for CWES

() * + / - cwe_likelihood

Additional Fields

safeguard_countermeasure

vulnerability

residual_risk

impact

threat

risk

complexity

severity

coupling

Generate The Formula

Prioritization Formula:

☐ Save Priority

Cancel

Run Priority

Alert Determination History

The ‘Previous’ field in the GUI stores the count of previous determinations, and if the hyperlinked number is selected, it brings up a comprehensive history of audit determinations for this meta-

alert including primary determinations, supplemental determinations, the Note field, and the Flag field. Auditors can browse all the old determinations per meta-alert, with this view.

Editing, Exporting, Sanitizing, or Deleting an Existing Project

First, select the Top Menu item "SCALE Analysis Tool", to get the SCALE homepage. It shows the list of active SCALE projects (see example screenshot below). For each project, icons (which have popup text when you hover the mouse over them) provide action options to:

- edit (the icon that looks like a pencil)
- export the project to a database (sqlite3) or a comma-separated value (CSV) file
- delete the project (the 'X' icon)

The screenshot shows the SCALE Analysis Tool interface. At the top is a dark navigation bar with the following items: a yellow circle icon, "SCALE Analysis Tool", "SCALE at CERT", "Help", and "Copyright (c) 2007-2018 Carnegie Mellon University". Below the navigation bar is the heading "Active SCALE Projects". Under this heading is a table with three columns: "Name", "Description", and "Project Options". The table lists three projects: "dos2unix", "project2", and "projectCoffee". Each project row has four icons in the "Project Options" column: a pencil (edit), a grid (export to CSV), a database cylinder (export to database), and an 'X' (delete). Below the table is a blue button labeled "New project". At the bottom of the page, there is a copyright notice: "SCALE Copyright (c) 2007-2018 Carnegie Mellon University. All Rights Reserved. See COPYRIGHT file for details."

Name	Description	Project Options
dos2unix		
project2		
projectCoffee		

[New project](#)

SCALE Copyright (c) 2007-2018 Carnegie Mellon University. All Rights Reserved. See COPYRIGHT file for details.

Exporting Analysis Results

After Eve is done auditing diagnostics, another auditor, Tom, can perform either a "table export" or a "database" export of the results of Eve's analysis. A table export is an archive of several files; each is a comma-separated value (CSV) file representing the audit data, including all the determinations. A database export is a SQLite3 database encoding the audit data, including all the determinations. Note that **a database export contains all the data from the project, but a table export does not** (e.g., a table export does not contain secondary diagnostic messages, may not contain all (or any) of the primary diagnostic messages, and may not include diagnostic IDs). Furthermore, a database export may be re-imported back into SCALE, while a table export cannot be

re-imported. An exported table archive is often further analyzed to produce statistics for a report. To obtain the table export, Tom goes to the SCALE web app homepage, logs in with the credentials supplied by Alice, and clicks the **Export Tables** icon for the desired project.

Tom saves the file to his local system. He can now open the file's CSV tables in a spreadsheet tool such as Excel. The CSV files contain information about every alert in the project, including the tool that produced the alert, the file and line number where the alert occurs, the message from the alert, the audit verdict, and so on.

Making a Sanitized Version of Exported Analysis Results

SCALE code includes a script that creates a "sanitized" version of an exported SCALE database, anonymizing fields (and deleting some fields) that may possibly contain sensitive data. In the current SCALE distribution, sanitizer functionality is provided only for exported SQLite3 databases, and sanitization can only be done via the command line.

Editing an Existing Project

Select the 'Edit' icon (looks like a pencil) for the project.

A screenshot of the edit screen is shown below. From this screen, you can add or replace tool outputs, modify the project name or description, upload determinations for the project, upload a SCALE database, or upload GNU Global pages. (Upload of a previously exported SCALE database may not be fully functional.)

Cascade Determinations

This feature allows a user to take determinations made from a previous SCALE audit and apply them to alerts generated by a new SCALE audit. It uses diff to determine if a code line within a file from a previous version of a codebase matches a code line within the current version of the codebase. If the lines are matched and there was a previously made determination for the meta-alert, then it infers the same determination for the current meta-alert. A new Note is added that identifies the verdict as being cascaded, with a timestamp (see another screenshot below).

NOTE OF CAUTION: Cascaded verdicts are not as trustworthy as direct verdicts, because data, control, and type flow changes may cause the previously correct determination to change. E.g., with different control flow, a previous True can become a currently correct False.

The section 'Upload Determinations from Project' in the 'Edit Project' page provides this functionality.

Edit project

Name:

Description:

[Upload Project](#)

[Upload SCALE Database](#)

[Browse...](#) No file selected. [Upload SCALE Database](#)

[Upload OSU Global Pages Archive \(.zip or .tgz\)](#)

[Browse...](#) No file selected. [Upload pages](#)

[Upload Determinations from Project](#)

[ondurix](#) [Upload determinations](#)

Upload new Diagnostics/Metric Test Outputs

Test & Platform	Test output	Script Output
☐ 11 /go /c	Browse... No file selected.	Success
☐ 12 /roscheckers /c	Browse... No file selected.	Success
☐ 21 /mmap /c	Browse... No file selected.	
☐ 22 /pdist /c	Browse... No file selected.	
☐ 23 /turkey /c	Browse... No file selected.	
☐ 24 /coveity /c	Browse... No file selected.	
☐ 25 /data /c	Browse... No file selected.	
☐ 26 /opcheck /c	Browse... No file selected.	Success
☐ 31 /wdtgen /java	Browse... No file selected.	
☐ 32 /findbugs /java	Browse... No file selected.	
☐ 33 /turkey /java	Browse... No file selected.	
☐ 34 /coveity /java	Browse... No file selected.	
☐ 35 /javadoc /java	Browse... No file selected.	
☐ 36 /findbugs /java	Browse... No file selected.	
☐ 41 /perror /perl	Browse... No file selected.	
☐ 42 /stat /perl	Browse... No file selected.	
☐ 53 /turkey /js	Browse... No file selected.	
☐ 61 /lizard /metric	Browse... No file selected.	
☐ 62 /oscan /metric	Browse... No file selected.	
☐ 63 /understand /metric	Browse... No file selected.	

[Update project database](#)

SCALE Copyright (c) 2007-2018 Carnegie Mellon University. All Rights Reserved. See LICENSE.txt for details.

After importing cascaded verdicts, auditor view shows Notes entries for cascaded determinations in the partial screenshot below:

Project: dos2unixForImportingDeterminations

[New Diagnostic](#)

Line:

Checker:

Sort direction:

Sort by:

Showing 1 to 10 of 238 | **Diagnostics per page:**

Set selected to:

☐	ID	Flag	Verdict	Supplemental	Notes	Pre
☐	721 (d)	[]	[False]	Edit	Cascaded from dos2unix on 2018-08-23_17:44:00	1
☐	719 (d)	[]	[True]	Edit	Cascaded from dos2unix on 2018-08-23_17:44:00	1
☐	720 (d)	[]	[True]	Edit	Cascaded from dos2unix on 2018-08-23_17:44:00	1
☐	734 (d)	[]	[Complex]	Edit	Cascaded from dos2unix on 2018-08-23_17:44:00	1
☐	735 (d)	[]	[Complex]	Edit	Cascaded from dos2unix on 2018-08-23_17:44:00	1
☐	736 (d)	[]	[Unknown]	Edit	0	0
☐	737 (d)	[]	[Unknown]	Edit	0	0
☐	738 (d)	[]	[Unknown]	Edit	0	0
☐	739 (d)	[]	[Unknown]	Edit	0	0
☐	790 (d)	[]	[Unknown]	Edit	0	0

15 Tools

15.1 Overview

Tools are an important part of code analysis. Since CREW can be used to analyze code bases of all ages and sizes, it is becoming increasingly important to incorporate as many tools as possible into the analysis. This provides additional context and information about the code, but also assists in scoping down complex code bases to individual instances that an analyst can then review. This is more feasible than spot-checking the entire codebase.

There are 2 types of tools that are used within CREW process: commercially available tools that are used unmodified, and internally developed analytic scripts.

While the CREW “Tools” tab only lists the commercially available tools that are used (see Section 15.2 below), other tools used within the rest of CREW are listed here as well, including the analytic scripts.

15.2 Tools

Below is a list (like that which would be included in the CREW “Tools” Tab), of tools used to complete the CREW analysis. **This list is subject to change, and SEI does not endorse any vendor or tool.**

TOOL	VENDOR	VERSION USED	UTILIZATION
Understand	Scientific Toolworks	6.2 Build 1125	Metrics, dependencies, code browser, CodeCheck SA
Unified Code Count (UCC)	Open source	2018.07	SLOC, complexity, maintainability index
PC-lint Plus	Vector	1.4.1	General code diagnostics
Parasoft C/C++test	Parasoft	2021.2.1	Security and general code diagnostics

Lattix Architect	Lattix	2022	Architecture and dependencies
GCC and Clang	Open source	Clang (15), GCC (N/A)	General code diagnostics
Sublime Text	Sublime	3.2.2	Code editor, search, browser
CppCheck	Open source	2.4.1	General Code Diagnostics
SCALe	SEI CERT	4.1.7	Static analysis tool for diagnostics aggregation and adjudication

15.3 Analytic Scripts

Below is a list of analytic scripts used through the CREW process:

SCRIPT NAME	PURPOSE/UTILIZATION
AssertionDensity.py	This script parses code files to look for and provide metrics on assertions.
ConditionalCompiles.py	This script parses code files to find and count the number of pre-compilation flags added to the code base.
DuplicateCodePostProcessing.py	This is a post processing script that takes the report generated by the perl script and provides code duplication metrics.

IncludesTogether.py	This script parses code files to find and count the number of includes that are not grouped together in the code base.
KeywordSearch.py	This script parses code files to find and count the number of different keywords in the code base.
KSFUsingUnderstand.py	This script uses the KSF to aid in code quality analysis and can additionally provide information on global variables.
MaintainabilityComments.py	This script parses code files to look for and provide metrics used for inputs into the maintainability comments CREW tabs.
FileHeaderAndCommentInfo.py	Analyze all files and get the distribution/metrics of file headers and comments. (Run through MaintainabilityComments.py)
GetTotalVariableDeclarations.py	Get the total number of global and local (file and function) declarations. (Run through MaintainabilityComments.py)
GetCommentsOnFunctionsAndClasses.py	Calculate metrics for classes, functions, and methods and preceding prologs. (Run through MaintainabilityComments.py)
CommentsPerFileDistribution.py	Get Top, Middle, and Bottom distribution of comments per file. (Run through MaintainabilityComments.py)
RedundantCode.py	This script parses code files to try to identify any lines of code that are redundant (directly repeated)

WhitespaceAnalysis.py

This script parses code files to do some whitespace analysis by looking at leading whitespace intervals and number of tabs.

16 Conclusion

Numerous high-profile cyberattacks on critical US computer systems emphasize the risks poor quality code presents to US national security and American economic leadership. Code Risk Estimation Worksheet (CREW) assessments were developed by SEI to help solve this problem, by identifying the *structural code quality*, e.g., the quality of a codebase, rather than *functional code quality*, e.g., how well the code fulfills mission requirements. CREW objectifies risk estimation by directing analysts to answer code-level questions after leveraging static analysis (SA) tools and reading subsections of the source code. This document serves as a user manual for CREW analysts and provides guidance on what methodologies to use when answering questions contained in CREW. The above chapters provide an overview of each CREW tab, followed by instructions on how to answer each individual question contained in the given tab. Ultimately, the CREW framework helps analysts improve the thoroughness, objectivity, consistency, and traceability of critical embedded system code assessments [Marchetti 2021].

17 CREW Manual Change Log

- v5.2.4:
 - Initial commit.
- v5.3.0:
 - Added unique identifiers to line items.
 - Added Change Log.
 - Updated line items in each tab based on internal review.

References/Bibliography

URLs are valid as of the publication date of this report.

[McConnell 2004]

Code Complete, Second Edition, by Steve McConnell, Microsoft Press, 2004.

[Bass 2013]

Software Architecture in Practice, Third Edition, by L. Bass, P. Clements, and R. Kazman, SEI Series in Software Engineering, Addison-Wesley, 2013.

[Koopman 2010]

Better Embedded System Software, by Philip Koopman, Drumnadrochit Press, 2010.

[Cois 2015]

“Continuous Integration in DevOps,” *SEI Blog*, by C. Aaron Cois, January 2015, <https://insights.sei.cmu.edu/devops/2015/01/continuous-integration-in-devops-1.html>

[Boehm 2007]

A SLOC Counting Standard, by Barry Boehm et al., Center for Systems and Software Engineering, USC, 2007, <http://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf>

[Booch 2008]

“Measuring Architectural Complexity,” by Grady Booch, *IEEE Software*, July 2008.
<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4548401>

[Serebrenik 2011]

2IS55 Software Evolution (2), by A. Serebrenik, Eindhoven University of Technology, 2011.
Software Engineering Concepts – Software Design Complexity, University of Houston,
http://scweb.sce.uhcl.edu/helm/SWEN-SoftwareEngineeringConcepts/my_files/TableContents/Module-10/software_design_complexity.html

[McCabe 1996]

Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication
500-235, T. McCabe et al., 1996. <http://www.mccabe.com/pdf/mccabe-nist235r.pdf>

[RAC 1996]

Introduction to Software Reliability: A State of the Art Review, Reliability Analysis Center (RAC), 1996.

[Tornhill 2015]

Your Code as a Crime Scene, by Adam Tornhill, Pragmatic Programmers, 2015.

[Google Engineering Tools 2011]

Bug Prediction at Google, Google Engineering Tools, <http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html>

[Feathers 2005]

Working Effectively with Legacy Code, by Michael Feathers, Prentice Hall, 2005.

[Pirsig 1974]

Zen and the Art of Motorcycle Maintenance, by Robert Pirsig, William Morrow, 1974.

[Lagerström 2018]

Exploring the Relationship Between Architecture Coupling and Software Vulnerabilities, Working Paper 18-031, Harvard Business School, by Robert Lagerström, et al., 2018.

[Ammann 2017]

Introduction to Software Testing, 2nd ed. P. Ammann, J. Offutt, Cambridge University Press, ISBN 978-1107172012, 2017.

[IEEE 1990]

IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

[ISSTA 1993]

Faults on its Sleeve - Amplifying Software Reliability Testing, Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA), Cambridge, Massachusetts, USA, June 1993.

[Marchetti 2021]

Assessing Software Quality Using a Risk-based Methodology, The Software Engineering Institute at Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, July 2021

[SEI 2022]

SCALe Collection, The Software Engineering Institute at Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, January 2023, <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=473847>

[Dewhurst 2022]

Static Code Analysis, Dewhurst, Ryan, The OWASP Foundation, January 2023, https://owasp.org/www-community/controls/Static_Code_Analysis

[Friedl 2006]

Mastering Regular Expressions, 3rd Edition, Friedl, Jeffrey, O'Reilly Media, Inc., ISBN: 9780596528126, August 2006

[CISQ 2022]

The Cost of Poor Software Quality in the US: A 2022 Report, Consortium for Information and Software Quality (CISQ), December 15, 2022

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 2023		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Code Risk Estimation Worksheet (CREW) v5.3.0 User Manual			5. FUNDING NUMBERS FA8702-15-D-0002	
6. AUTHOR(S)				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER Error! No text of specified style in document.	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100			10. SPONSORING/MONITORING AGENCY REPORT NUMBER n/a	
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS)				
14. SUBJECT TERMS			15. NUMBER OF PAGES 225	
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

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