CISQ Conformance Assessment Method for Software Certification Technology

Based on CISQ's Automated Source Code Quality Measures

A Guide for Assessing the Level of Conformance of a Static Analysis Product in Detecting the Weaknesses Comprising the CISQ Automated Source Code Quality Measures

CISQ–TR–2021–01

Consortium for Information and Software Quality

Executive Summary

This document presents the method for assessing a static analysis product's conformance to the four CISQ Automated Source Code Quality Measures (Security, Reliability, Performance Efficiency, and Maintainability). These measures are currently OMG approved standards and will be published as ISO standards in 2021. These measures assess the structural quality of a software system's source code, a primary source of risk affecting an application's cybersecurity, operational performance, and cost of ownership. Each measure is composed of a list of severe weaknesses that violate rules of good architectural and coding practice. These weaknesses can be detected using static analysis technology provided by commercial vendors. All weaknesses are listed in the Common Weakness Enumeration Repository with a CWE identifier. Each measure is developed by counting the occurrences of relevant weaknesses in the source code. These counts can then be divided by a size measure to determine the to develop a six-sigma rating for each of the four quality characteristics. These results can be used to certify the level of quality in a specific release or version of a software system. The analysis also identifies severe weaknesses that are not allowed to exist in the code. The evidence required to demonstrate a technology's ability to detect and measure the structural quality violations is listed for each of the four CISQ measures. Forms are provided for capturing the evidence and level of compliance, represented as the extent to which a technology detects and measures all the CISQ weaknesses and the various ways they are instantiated in an application's source code.

CISQ has worked closely with **IT-AAC** in publicizing and deploying these measurement standards, and in preparing them for use in software acquisition.

Table of Contents

1. Structural Quality Certification

1.1 What Is a Structural Quality Certification?

Structural quality certification provides managers, executives, and customers with a quantitative indicator of the risk and cost associated with a software system. Therefore, a set of certifications covering an organization's business or mission critical software-intensive systems provides evidence that the risks and costs of these applications are under governance. Each certification is based on the static analysis of the system's source code to detect weaknesses that violate rules of good architectural and coding practice. The results are presented as measures for each of four quality characteristics—Security, Reliability, Performance Efficiency, and Maintainability. These analyses and measures can be used as:

- an indicator of the business/mission risk or maintenance cost associated with an application,
- a guide for remediating weaknesses that will reduce the risk of cost of an application,
- a basis for tracking trends in the risk or cost of an application over time, and
- a sign that the risks and costs of applications are being governed and managed.

CISQ does not certify applications. Rather, the level of quality in an application can be certified by a CISQ conformant service provider using a CISQ conformant static analysis technology. CISQ conformant service providers will present customers with a certificate that expresses an application's quality level on one or more CISQ measures in density or sigma form (weaknesses per million opportunities, to be described later).

The CISQ certification is an indicator rather than an absolute measure of application quality for two reasons.

1. The CISQ measures do not include all weaknesses related to a specific quality characteristic, only those that were deemed severe enough to require remediation. However, a measure constructed on these weaknesses should be a strong correlate of the amount of less severe weaknesses that were not measured.
2. A CISQ quality measure is only valid until the next patch is implemented on the code. Consequently, the injection of new weaknesses through enhancements, modifications, or deletions will change the certification measurements. However, since the results are reported as sigma levels the actual quality levels may not change dramatically after standard maintenance activities, especially if good quality assurance activities are applied.

Consequently, a CISQ certification does not guarantee a level of defect-freeness or a level of operational performance. However, it correlates highly with these outcomes and provides management with insight into the risks and costs associated with an application. It also provides evidence that management is governing these risks and costs when unacceptable certification results lead to remedial actions.

1.2 What Are the CISQ Structural Quality Measures?

The four CISQ Automated Source Code Quality Measures were selected by executives from the companies who formed CISQ. These 4 measures were selected from among 10 candidates as being the most relevant to their application challenges in 2010. Teams of experts from the original 24 companies

that formed CISQ met over a 2-year period to select the weaknesses that would constitute each of the four measures. A weakness was only included in a CISQ measure if a majority of software professionals would agree it had to be remediated. However, the Security measure weaknesses were taken from the top 25 CWEs, OWASP Top 10, and related security lists. For instance, the Security measure includes such weaknesses as SQL injection, cross-site scripting, and buffer overflows.

In 2019 these four measures were updated to include weaknesses relevant to embedded systems to provide broad coverage of software types. All four were collected into a single standard called Automated Source Code Quality Measures (https://www.omg.org/spec/ASCQM/1.0). All weaknesses constituting these measures are listed in the Common Weakness Enumeration Repository maintained by MITRE Corporation and are labelled with CWE identifiers. The four CISQ Quality Characteristic measures include:

- **Security**—a measure of the extent to which software contains weaknesses that can be exploited to gain unauthorized access to a system to steal data, cause damage, or other malicious acts.
- **Reliability**—a measure of the extent to which software contains weaknesses that cause outages, unexpected behavior, instability, data corruption, long recovery times, or other related problems.
- **Performance Efficiency**-- a measure of the extent to which software contains weaknesses that can degrade a system's performance or cause excessive use of processor, memory, or other resources.
- **Maintainability--** a measure of the extent to which software contains weaknesses that make software hard to understand or change, resulting in excessive maintenance time and cost as well as higher defect injection rates.

1.3 Why Are the CISQ Quality Measures Valuable for Certifications

The CISQ Automated Source Code Quality Measures are an international standard supported by OMG (https://www.omg.org/spec/ASCQM/1.0). They have been approved to become an ISO standard and expect to receive publication approval as ISO/IEC 5055 in early 2021 (https://www.iso.org/standard/80623.html). This is the only standard that assesses quality at the level of weaknesses in the source code at both the architectural and code levels. Consequently, the CISQ measures provide that following benefits for certifications.

1. CISQ Quality Measures have an established and well-defined definition, therefore they provide a common language for customers, suppliers, auditors, contract officials, and others for communicating about application quality issues and expectations.
2. CISQ Quality Measures provide a common basis for benchmarking quality across systems, technologies, vendors, and companies. They provide a rigorous basis for developing baselines and thresholds that are specific to technology, industry vertical, application type and other demographic characteristics.
3. CISQ Quality Measures are directly related to the weaknesses in software that create risk and cost. Therefore, they provide leading indicators of potential operating problems or excessive maintenance costs.

1.4 How Do CISQ Measures Relate to ISO Standards?

The definition and coverage of CISQ Quality Measures are conformant with the definitions of software quality characteristics and sub-characteristics in ISO/IEC 25010, which replaces the old ISO/IEC 9126. ISO/IEC 25010 defines a quality characteristic as being composed from several quality sub-characteristics. This framework for software product quality is presented in Figure 1 for the eight quality

characteristics presented in 25010. The quality characteristics and their sub-characteristics selected for source code measurement by CISQ are indicated in blue.

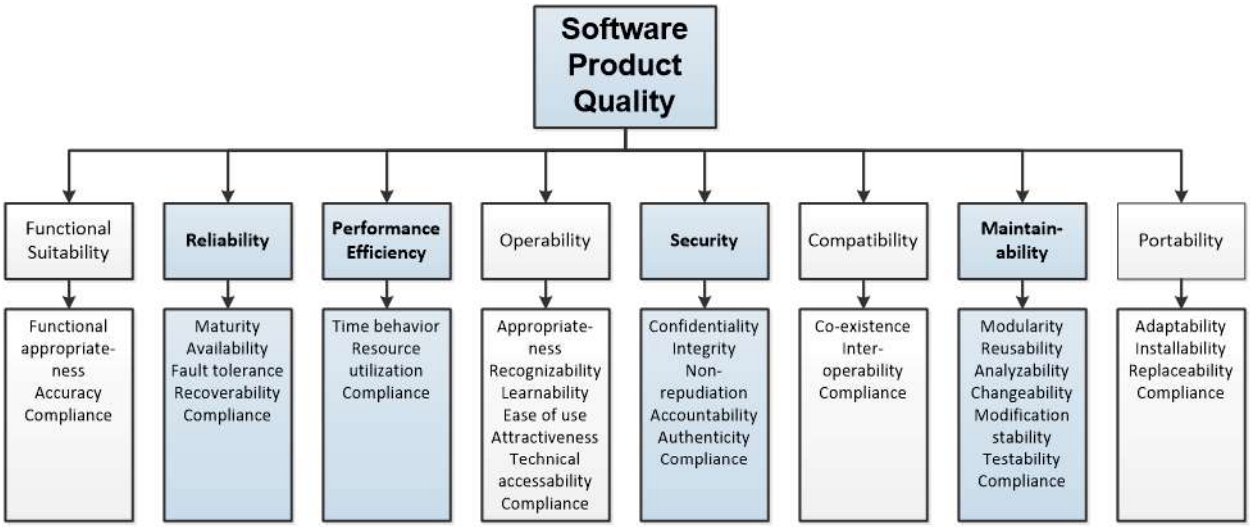

Figure 1. Software Quality Characteristics from ISO/IEC 25010 with CISQ focal areas highlighted

The definitions of actual software quality measures are provided in ISO/IEC 25023 where each quality characteristic is quantified by a collection of measurable attributes of software, such as the violation of a quality rule. However, the measures provided in ISO/IEC 25023 are generally defined at the level of system behavior and do not measure the actual source code, that is they are not based on measuring the software weaknesses that cause unacceptable system behaviors. The CISQ measures supplement ISO/IEC 25023 by providing measures of quality-related weaknesses in the source code.

1.5 How Are Certification Scores Presented to Customers?

A CISQ certification is presented in the Sigma format that many companies are familiar with from Six Sigma quality improvement programs. The use of Sigma levels provides a common representation supported by a rigorous, statistically-based method for benchmarking quality results. A Sigma (σ) is a standard deviation from the mean score of a distribution. In quality programs Sigma results are interpreted as frequencies of occurrence based on the Normal Distribution. For instance, a score of 6σ is six standard deviations from the mean, and indicates no more than 4 defects per million points of inspection. A point of inspection is interpreted as opportunity for a defect to have occurred and been detected. In measuring quality, the Sigma level is determined by the number of weaknesses per million opportunities. The fewer weaknesses per million opportunities, the higher the Sigma score.

In software, opportunities are measured by the number of times an architectural or coding rule could be applied to a construct in the source code. Thus, the CISQ Certification score on one of the four measures applied to an application is the number of occurrences of each weakness included in the CISQ measure compared to the number of times the CISQ rule violated by the weakness could be applied to constructs in the source code. For instance, if there were 2000 modules in an application then there would 2000 opportunities to violate a rule regarding excessive coupling. The number of weaknesses per million opportunities declines exponentially as the Sigma level increases.

The total number of occurrences detected for the weaknesses included in a CISQ measure is then transformed into weaknesses per million opportunities to determine the Sigma level for that measure. The thresholds for each Sigma level are as follows.

- 1σ — 697,672 weaknesses per million opportunities — 69.7672 % defective
- 2σ — 308,537 weaknesses per million opportunities — 30.8537 % defective
- 3σ — 66,807 weaknesses per million opportunities — 6.6807 % defective
- 4σ — 6,210 weaknesses per million opportunities — .6210 % defective
- 5σ — 233 weaknesses per million opportunities — .0233 % defective
- 6σ — 3.4 weaknesses per million opportunities — .0003% defective

In general applications, a score below the 3 Sigma level should be considered unacceptable and high risk. In practice, it will be difficult for applications to achieve 5 or 6 Sigma scores, and this level of quality may be beyond the requirements for many applications, or beyond the cost-benefits of striving for this level of quality. However, there are violations such as SQL injection for which the tolerance level should be '0 occurrences' since the Security risks posed by this weakness can be disastrous. The appropriate quality range for most business applications will be a certification between 3 and 4 Sigma. Figure 2 presents an example of certificate reporting Sigma levels for an application that would be presented to customers.

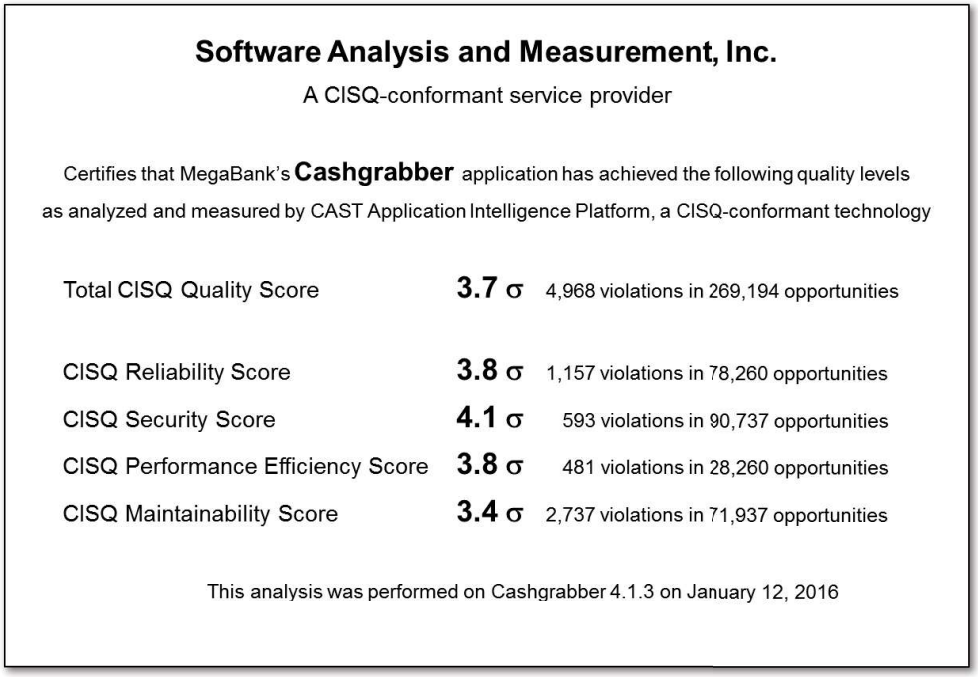

Figure 2. Example Certification Report

Other quality measures can be added to the certification such as the density of weaknesses based on the size of the application. In fact, density measures may be more appropriate than Sigma notations for small applications (<10,000 lines of source code) because of the constrained number of opportunities where architectural and coding rules can be applied.

2. The CISQ Conformance Assessment for Technology

2.1 What Is the CISQ Conformance Assessment Process?

An endorsement for CISQ conformance is awarded by a CISQ-authorized Lead Appraiser to a vendor's source code analysis and measurement technology. The Appraiser will evaluate the vendor's technology either at the vendor's site, or under agreement at another site, to assess the evidence of CISQ-conformance. The CISQ Conformance Assessment is conducted under a non-disclosure agreement with the vendor. The vendor may seek a conformance endorsement for all four CISQ Quality Measures or a subset of the measures.

Detection patterns can differ among programming languages and related technologies. Therefore, a conformance assessment must be performed separately for each programming language or related technology the vendor's technology supports that would be included in a CISQ-based certification. Vendors shall clearly indicate the programming languages and related technologies for which their technology has been endorsed as CISQ-conformant.

There are five steps in the CISQ Conformance Assessment process.

1. The Appraiser will review an evidence-based claim that they detect the weaknesses in a CISQ measure as demonstrated by the output from their analyses of customer applications that their technology can detect the weaknesses comprising the CISQ measures.
2. The Appraiser will review descriptions of how the technology detects and counts weaknesses. The vendor should not reveal any trade secrets in describing their analysis procedures. Nevertheless, the vendor should describe analytic procedures in sufficient detail that the Appraiser can determine whether they are effective approaches to detecting the weakness. If necessary, and both parties agree, the Appraiser can evaluate analytic techniques in the source code of the technology under evaluation.
3. The Appraiser will interview several members of the vendor's staff to affirm the information in the conformance case. Typical interviewees might include system designers, software developers, technology installers or operators, solution consultants, and a technology manager.
4. The Appraiser should observe a customer analysis to determine verify information in the case and to determine any limits on the analysis technology such as application size or multiple languages. If a customer analysis is not available, the Appraiser should observe an analysis on a vendor testbed. The applications analyzed should be of sufficient size to demonstrate the capability of the technology and validity of the conformance case.
5. The Appraiser will prepare a report of findings regarding conformance regarding the detection of each weakness for each measure in the scope of the assessment. This report will be submitted both to the vendor and the CISQ office. Optionally the Appraiser can help the vendor revise their conformance case for presentation to customers based on the evidence and limitations observed. The conformance assessment will normally take between one and three days.

The Appraiser will develop an agreement with the vendor on the scope and process steps of the CISQ Conformance Assessment. This agreement will be documented in an assessment plan which can be incorporated into an assessment contract or other formal agreement. At a minimum, the plan will include the following information.

- The CISQ Measures to be assessed for conformance

- The dates and estimated durations for each of the five tasks in the assessment process
- The analysis reports to be evaluated and the analyses to be observed
- The number and roles of employees to be interviewed
- The data for submitting a final conformance report

2.2 What Is Required to Demonstrate Conformance?

Vendors of technology wanting to be designated as 'CISQ Conformant' must present an evidence-based case to support their claim of conformance. The conformance claim and supporting arguments backed by evidence that justifies the validity of the claim will be evaluated by a CISQ-authorized assessor to determine if the case is sufficient to support the conformance claim. The assessor will also want to see the technology used in an analysis. The vendor must be able to demonstrate the following four capabilities.

1. *Capability to Load and Prepare Software for Analysis*—The vendor shall maintain an auditable description of the process for preparing the software for the analysis. The description shall include a list of all information and artifacts required to configure the software and support its analysis. The process description shall describe the tasks for obtaining the software and related information, ensuring the protection of customer confidential information and intellectual property, loading the software, and configuring the software for analysis.
2. *Capability to Analyze and Detect CISQ Measure Weaknesses*—The vendor shall maintain an auditable description of how its technology analyzes each of the violations in each CISQ Quality Characteristic measure for which conformance is being assessed. This description shall include methods for detecting violations both within and across code units, modules, and components where violations can involve multiple program elements.
3. *Capability to Accurately Compute CISQ Quality Characteristic Measures*—The vendor shall maintain an auditable description of how detected violations are counted and aggregated into a CISQ-conformant score for each CISQ Measure which is being assessed for conformance. If other measures such weakness density are being reported as part of the certification, the auditable description shall include them.
4. *Capability to Provide Auditable Reports of Results*—The vendor's analysis and measurement technology shall provide auditable reports that include the number and location of violations for each of the CISQ violations included in each CISQ Measure being assessed. These reports will include all measures computed from the detected weaknesses. These reports should be standard outputs that are presented to customers.

Claims of conformance will be made individually for each CISQ measure, and sub-claims of conformance will be made for each weakness included in the measure. In the case where a weakness may have several different instantiations each requiring a different mode of detection, a third tier of claims must be made regarding these different weakness modes.

Some vendor technologies may be unable to detect all the weaknesses comprising each CISQ measure at the initiation of the CISQ Conformance program. Consequently, CISQ has established a sliding scale of requirements for conformance during the period of 2017 to 2019. During these years the number of weaknesses that must be detected to achieve conformance will gradually increase as depicted in Table 1. By the end of 2019, vendor technologies must not fail to detect more than one of the weaknesses included in a measure in to be conformant with that measure.

CISQ Measures	# weaknesses for conformance			Total weaknesses per measure
	2021	2022	2023	
Security	62	68	73	73
Reliability	64	70	74	74
Perform. Eff.	15	17	19	19
Maintainability	25	27	29	29

Table 1. Yearly Requirement for Number of Weaknesses Detected to Achieve Conformance

The required percentages are more lenient for Reliability and Security, reflecting the greater analytic difficulty of detecting weaknesses in these measures. This graduated compliance scheme allows vendors sufficient time to develop the analytic and measurement capabilities required for conformance. In their evidence-based conformance case are required to indicate what weaknesses they are unable to detect, or any instantiations of a weakness they are unable to detect.

2.3 How Is an Evidence-Base Case Constructed?

Each claim is associated with an argument that the claim is true. Arguments are supported by evidence that demonstrate the validity of the claim. An argument states the kind of evidence required and why that demonstrates the truth of the claim. Thus, the evidence must support the argument and be sufficient to judge its truth. The hierarchy of conformance claims in presented in Figure 3.

Validated subclaims can be used to provide evidence for the validity of higher-level claims. That is, the fact that the weaknesses comprising a measure can be detected and counted provides evidence that argues the technology is CISQ-conformant for that measure.

The hierarchy of claims in a CISQ conformance case for technology compliance will include the first two and possibly the third level of claims as follows:

- *Measure Level*—a claim that the Technology achieves the level of detection required for conformance for a specific measure. The level of detection will be determined by verified claims regarding each of the weaknesses incorporated into a CISQ measure. During the initial three years of the CISQ Conformance Assessment program, the level of compliance may be different for each CISQ measure based on the difficulty of detecting the various weaknesses incorporated into it.
- *Weakness Level*—a claim that a weakness can be detected by the technology in one or more of its instantiations in code. The claim for a weakness may be verified for some instantiations, but not others. In which case the claim will be conditional and undetectable instantiations must be declared.
- *Instantiation Level*—a weakness may occur in several different forms, or instantiations, in the code. Each instantiation must be verified with evidence that supports the argument for that instantiation. Instantiations that cannot be accurately detected must be declared and represented in the evidence-based case presented to the CISQ Assessor. Undetected instantiations must also be declared in materials presented to customers so they know the limits

of the certification assessments performed. If has only a single instantiation, then no third-tier claims need to be made for the weakness.

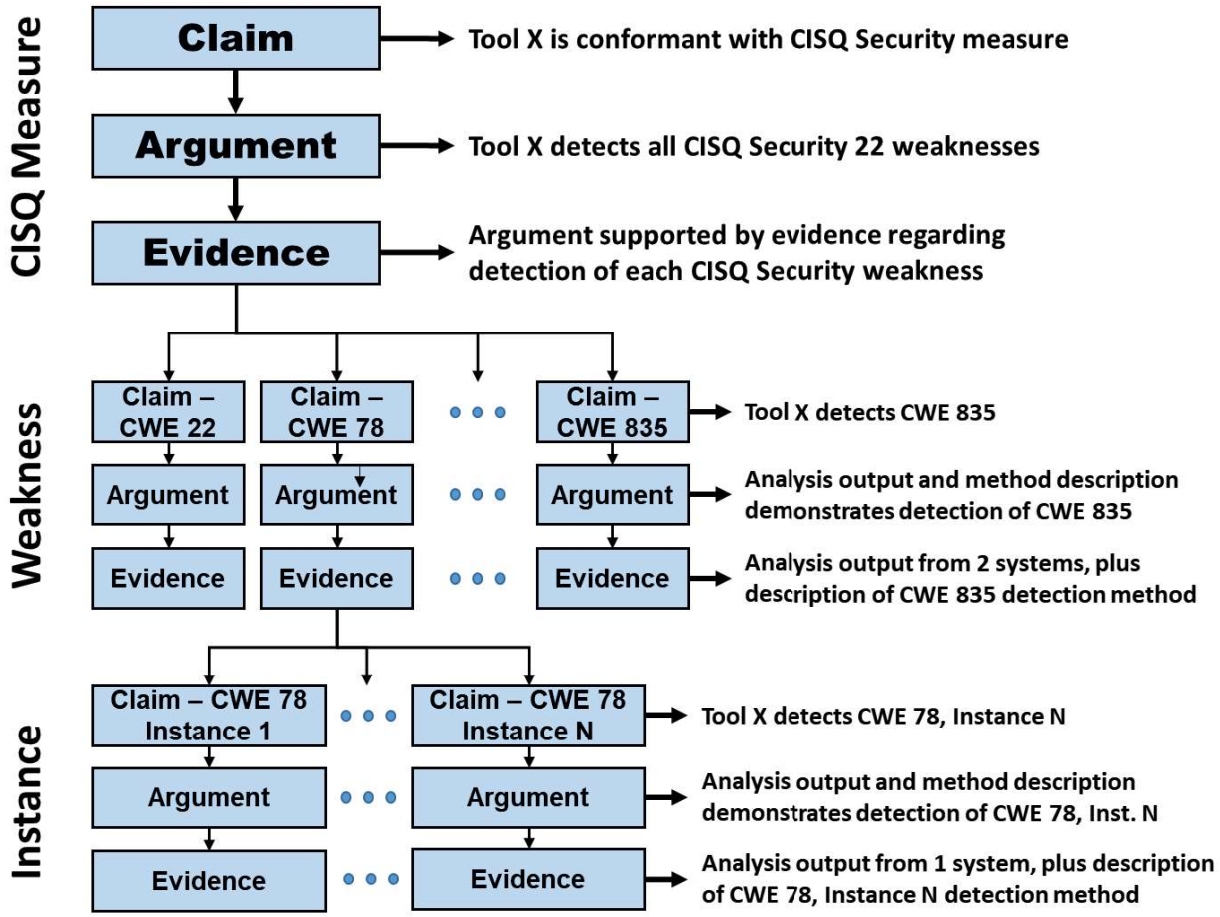

Figure 3. The Hierarchy of Claims for a CISQ Conformant Technology

2.4 What Evidence Is Required?

Claims and arguments for CISQ conformance must be supported by several forms of evidence for each of the CISQ measures to which the vendor claims conformance. The primary types of evidence to be presented in a CISQ consist of the following.

- Output from one or more automated structural quality analyses
- Descriptions of how each CISQ weakness is detected by the technology
- Onsite interviews with developers and operators of the technology
- Observation of an actual structural quality analysis

Evidence of detection-—The most important form of evidence is output from the results of a structural quality analysis. This form of evidence is required and guidance for its content can be found in Section 3. The analysis output must provide the following information.

1. A list of CISQ weaknesses detected with their CISQ identifiers
2. The count of weaknesses detected for each type of CISQ weakness
3. The total of all CISQ weaknesses detected
4. The location in the code of each CISQ weakness detected

Evidence from detection method—Complementing evidence of detection from the output is a description of the analysis methods used to detect each of the CISQ weaknesses. Method descriptions are a requirement in the assessment process. The method used should be described in enough detail for a knowledgeable assessor to evaluate its reasonableness, but proprietary or trade secret information should not be revealed. A high-level description of methods should also be included in the documented conformance case. This form of evidence should include the following information.

1. The method for loading and preparing the software for analysis
2. How the code pattern representing each potential weakness is identified
3. How the elements of each pattern relevant to the weakness are analyzed
4. In the case of system-level weaknesses, how patterns and elements are analyzed across applications layers
5. How the weakness is determined and distinguished from a false positive
6. In the case of weaknesses with multiple instantiations, how each instantiation is identified, analyzed, and distinguished from a false positive
7. A list of instantiations that cannot be detected

Evidence from interviews—Assertions from interviews with developers or operators can be used to verify the evidence collected from analysis outputs and method descriptions. Developers can be questioned about the methods used in identifying patterns and detecting weaknesses. Operators can be questioned about how application is prepared for analysis and the types of problems have been encountered during analysis. The purpose of interviewing the vendor's staff include the following.

- Verifying and affirming evidence from outputs and method descriptions
- Clearing up points of confusion
- Assisting the vendor in preparing their conformance case for presentation to customers
- Recommendations for improvement from the assessor's experience

In addition to the evidence regarding weaknesses enumerated above, the method for calculating the Sigma level and other measures provided in the certification must also be inspected. Evidence regarding the calculation of the Sigma level should include the following.

1. The method for counting weaknesses and aggregating scores
2. The transformation of the scores into weaknesses per million opportunities
3. The translation of weaknesses per million opportunities into Sigma notation based on a normal distribution
4. Methods for calculating any other measures provided in the certification such as weakness density.

The evidence should be organized in a conformance case that can be evaluated by a CISQ Assessor prior to an on-site visit. Forms for recording evidence are provided in Appendix A.

2.5 The Endorsement of CISQ Conformance

The CISQ-authorized Assessor will determine the level of conformance of the vendor's analysis and measurement technology. If the vendor's technology does not satisfy the required threshold of conformance for any of the CISQ measures within scope of the assessment, the Assessor will describe any necessary remediation actions to achieve the threshold required for endorsement as CISQ-conformant. For each CISQ Measure for which the threshold for conformance has been achieved, the Assessor will provide the vendor an Endorsement of CISQ Conformance. The vendor is then entitled to advertise conformance for those specific CISQ measures.

3. What Is Required for Each Structural Quality Measure

3.1 Assessment Guidance for Evaluating Weakness Detection

CISQ-authorized Assessors will evaluate the capability of a technology to detect and analyze patterns in software where rules of good architectural and coding practice can be violated. A technology must be able to identify the program elements incorporated into the pattern and detect the anomaly in their structure, organization, relationships, or interaction that creates the CISQ weakness. The following tables present the structures that should be detected for each weakness in each CISQ measure in order to argue the claim that a weakness can be detected. How these elements are detected would be provided in the evidence regarding the analysis method. The columns in these tables provide the following information.

- *Weakness identifier*—the descriptive name of the weakness
- *Detection Level*—indicated in yellow lettering directly below the weakness identifier, the detection level indicates program levels that may contain structural elements of the weakness, and which therefore must be analyzed to detect the weakness. A weakness can be limited to only one level, or it can incorporate all three. The three levels include
    - *Unit*—a self-contained group of computer instructions such as a class, method, module, sub-routine, etc., often separately compiled
    - *Technology*—a collection of code units that form a sub-system, layer, or some other significant grouping, and are written in the same language
    - *System*—the full collection of technology groupings across the various layers of an application that form its architecture and software content
- *Descriptor*—a short name for the weakness, chosen from common usage where possible.
- *Detection pattern*—a description of the weakness pattern that includes the source code elements that must be detected to identify the weakness during automated code analysis.

The following sections list the weaknesses included in each of the four CISQ Quality Measures. Each weakness is listed with it CWE identifier, descriptor title, and textual description of the weakness. Some weaknesses are listed as parents, indicated by the CWE number being presented in a dark blue cell. These parent weaknesses have various instantiations that are presented as child or contributing weaknesses that are indicated by their CWE identifier being presented in light blue cells.

3.2 Security

Security measures the extent to which software contains weaknesses that can be exploited to gain unauthorized access to a system to steal data, cause damage, or other malicious acts. The quality weaknesses composing the CISQ Automated Source Code Security Measure are presented in Table 2. This measure contains 36 parent weaknesses and 37 contributing weaknesses (children in the CWE) that represent variants of these weaknesses. The CWE numbers for contributing weaknesses are presented in light blue cells immediately below the parent weakness whose CWE number is in a dark blue cell.

Table 2. Weaknesses composing the Automated Source Code Security Measure

CWE #	Descriptor	Weakness description
CWE-22	**Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')**	The software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.
CWE-23	**Relative Path Traversal**	The software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory.
CWE-36	**Absolute Path Traversal**	The software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize absolute path sequences such as "/abs/path" that can resolve to a location that is outside of that directory.
CWE-77	**Improper Neutralization of Special Elements used in a Command ('Command Injection')**	The software constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component.
CWE-78	**Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')**	The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.
CWE-88	**Argument Injection or Modification**	The software does not sufficiently delimit the arguments being passed to a component in another control sphere, allowing alternate arguments to be provided, leading to potentially security-relevant changes.

CWE-79	**Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**	The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users. Cross-site scripting (XSS) vulnerabilities occur when: 1. Untrusted data enters a web application, typically from a web request. 2. The web application dynamically generates a web page that contains this untrusted data. 3. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc. 4. A victim visits the generated web page through a web browser, which contains malicious script that was injected using the untrusted data. 5. Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain. 6. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.
CWE-89	**Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**	The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.
CWE-564	**SQL Injection: Hibernate**	Using Hibernate to execute a dynamic SQL statement built with user-controlled input can allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.
CWE-90	**Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')**	The software constructs all or part of an LDAP query using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended LDAP query when it is sent to a downstream component.
CWE-91	**XML Injection (aka Blind XPath Injection)**	The software does not properly neutralize special elements that are used in XML, allowing attackers to modify the syntax, content, or commands of the XML before it is processed by an end system.
CWE-99	**Improper Control of Resource Identifiers ('Resource injection')**	The software receives input from an upstream component, but it does not restrict or incorrectly restricts the input before it is used as an identifier for a resource that may be outside the intended sphere of control.

CWE-119	**Improper Restriction of Operations within the Bounds of a Memory Buffer**	The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.
CWE-120	**Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')**	The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.
CWE-123	**Write-what-where condition**	Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.
CWE-125	**Out-of-bounds Read**	The software reads data past the end, or before the beginning, of the intended buffer.
CWE-130	**Improper Handling of Length Parameter Inconsistency**	The software parses a formatted message or structure, but it does not handle or incorrectly handles a length field that is inconsistent with the actual length of the associated data.
CWE-786	**Access of Memory Location Before Start of Buffer**	The software reads or writes to a buffer using an index or pointer that references a memory location prior to the beginning of the buffer. This typically occurs when a pointer or its index is decremented to a position before the buffer, when pointer arithmetic results in a position before the beginning of the valid memory location, or when a negative index is used.
CWE-787	**Out-of-bounds Write**	The software writes data past the end, or before the beginning, of the intended buffer. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer.
CWE-788	**Access of Memory Location After End of Buffer**	The software reads or writes to a buffer using an index or pointer that references a memory location after the end of the buffer. This typically occurs when a pointer or its index is decremented to a position before the buffer; when pointer arithmetic results in a position before the buffer; or when a negative index is used, which generates a position before the buffer.
CWE-805	**Buffer Access with Incorrect Length Value**	The software uses a sequential operation to read or write a buffer, but it uses an incorrect length value that causes it to access memory that is outside of the bounds of the buffer.

CWE-822	**Untrusted Pointer Dereference**	The program obtains a value from an untrusted source, converts this value to a pointer, and dereferences the resulting pointer. There are several variants of this weakness, including but not necessarily limited to: • The untrusted value is directly invoked as a function call. • In OS kernels or drivers where there is a boundary between "userland" and privileged memory spaces, an untrusted pointer might enter through an API or system call (see CWE-781 for one such example). • Inadvertently accepting the value from an untrusted control sphere when it did not have to be accepted as input at all. This might occur when the code was originally developed to be run by a single user in a non-networked environment, and the code is then ported to or otherwise exposed to a networked environment.
CWE-823	**Use of Out-of-range Pointer Offset**	The program performs pointer arithmetic on a valid pointer, but it uses an offset that can point outside of the intended range of valid memory locations for the resulting pointer. • While a pointer can contain a reference to any arbitrary memory location, a program typically only intends to use the pointer to access limited portions of memory, such as contiguous memory used to access an individual array. • Programs may use offsets to access fields or sub-elements stored within structured data. The offset might be out-of-range if it comes from an untrusted source, is the result of an incorrect calculation, or occurs because of another error.
CWE-824	**Access of Uninitialized Pointer**	The program accesses or uses a pointer that has not been initialized. If the pointer contains an uninitialized value, then the value might not point to a valid memory location.
CWE-825	**Expired Pointer Dereference**	The program dereferences a pointer that contains a location for memory that was previously valid, but is no longer valid.
CWE-129	**Improper Validation of Array Index**	The product uses untrusted input when calculating or using an array index, but the product does not validate or incorrectly validates the index to ensure the index references a valid position within the array.
CWE-134	**Use of Externally Controlled Format String**	The software uses a function that accepts a format string as an argument, but the format string originates from an external source.
CWE-252	**Unchecked Return Value**	The software does not check the return value from a method or function, which can prevent it from detecting unexpected states and conditions.

CWE-404	**Improper Resource Shutdown or Release**	The program does not release or incorrectly releases a resource before it is made available for re-use.
CWE-401	**Improper Release of Memory Before Removing Last Reference ('Memory Leak')**	The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.
CWE-772	**Missing Release of Resource after Effective Lifetime**	The software does not release a resource after its effective lifetime has ended, i.e., after the resource is no longer needed.
CWE-775	**Missing Release of File Descriptor or Handle after Effective Lifetime**	The software does not release a file descriptor or handle after its effective lifetime has ended, i.e., after the file descriptor/handle is no longer needed. When a file descriptor or handle is not released after use (typically by explicitly closing it), attackers can cause a denial of service by consuming all available file descriptors/handles, or otherwise preventing other system processes from obtaining their own file descriptors/handles.
CWE-424	**Improper Protection of Alternate Path**	The product does not sufficiently protect all possible paths that a user can take to access restricted functionality or resources. When data storage relies on a DBMS, special care shall be given to secure all data accesses and ensure data integrity.
CWE-434	**Unrestricted Upload of File with Dangerous Type**	The software allows the upload or transfer files of dangerous types that can be automatically processed within the product's environment.
CWE-477	**Use of Obsolete Function**	The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained.
CWE-480	**Use of Incorrect Operator**	The programmer accidentally uses the wrong operator, which changes the application logic in security-relevant ways.
CWE-502	**Deserialization of Untrusted Data**	The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.
CWE-570	**Expression is Always False**	The software contains an expression that will always evaluate to false.
CWE-571	**Expression Is Always True**	The software contains an expression that will always evaluate to true.
CWE-606	**Unchecked Input for Loop Condition**	The product does not properly check inputs that are used for loop conditions, potentially leading to a denial of service because of excessive looping.
CWE-611	**Improper Restriction of XML External Entity Reference ('XXE')**	The software processes an XML document that can contain XML entities with URIs that resolve to documents outside of the intended sphere of control, causing the product to embed incorrect documents into its output.

CWE-643	**Improper Neutralization of Data within XPath Expressions ('XPath Injection')**	The software uses external input to dynamically construct an XPath expression used to retrieve data from an XML database, but it does not neutralize or incorrectly neutralizes that input. This allows an attacker to control the structure of the query.
CWE-652	**CWE-652 Improper Neutralization of Data within XQuery Expressions ('XQuery Injection')**	The software uses external input to dynamically construct an XQuery expression used to retrieve data from an XML database, but it does not neutralize or incorrectly neutralizes that input. This allows an attacker to control the structure of the query.
CWE-665	**Improper Initialization**	The software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state when it is accessed or used.
CWE-456	**Missing Initialization of a Variable**	The software does not initialize critical variables, which causes the execution environment to use unexpected values.
CWE-457	**Use of uninitialized variable**	The software uses a variable that has not been initialized leading to unpredictable or unintended results.
CWE-662	**Improper Synchronization**	The software attempts to use a shared resource in an exclusive manner, but does not prevent or incorrectly prevents use of the resource by another thread or process.
CWE-366	**Race Condition within a Thread**	If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.
CWE-543	**Use of Singleton Pattern Without Synchronization in a Multithreaded Context**	The software uses the singleton pattern when creating a resource within a multithreaded environment.
CWE-567	**Unsynchronized Access to Shared Data in a Multithreaded Context**	The product does not properly synchronize shared data, such as static variables across threads, which can lead to undefined behavior and unpredictable data changes.
CWE-667	**Improper Locking**	The software does not properly acquire a lock on a resource, or it does not properly release a lock on a resource, leading to unexpected resource state changes and behaviors.
CWE-820	**Missing Synchronization**	The software utilizes a shared resource in a concurrent manner but does not attempt to synchronize access to the resource.
CWE-821	**Incorrect Synchronization**	The software utilizes a shared resource in a concurrent manner but it does not correctly synchronize access to the resource.
CWE-672	**Operation on a Resource after Expiration or Release**	The software uses, accesses, or otherwise operates on a resource after that resource has been expired, released, or revoked.

CWE-415	**Double Free**	The product calls free() twice on the same memory address, potentially leading to modification of unexpected memory locations.
CWE-416	**Use After Free**	Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.
CWE-681	**Incorrect Conversion between Numeric Types**	When converting from one data type to another, such as long to integer, data can be omitted or translated in a way that produces unexpected values. If the resulting values are used in a sensitive context, then dangerous behaviors may occur. For instance, if the software declares a variable, field, member, etc. with a numeric type, and then updates it with a value from a second numeric type that is incompatible with the first numeric type.
CWE-194	**Unexpected Sign Extension**	The software performs an operation on a number that causes it to be sign-extended when it is transformed into a larger data type. When the original number is negative, this can produce unexpected values that lead to resultant weaknesses.
CWE-195	**Signed to Unsigned Conversion Error**	The software uses a signed primitive and performs a cast to an unsigned primitive, which can produce an unexpected value if the value of the signed primitive cannot be represented using an unsigned primitive.
CWE-196	**Unsigned to Signed Conversion Error**	The software uses an unsigned primitive and performs a cast to a signed primitive, which can produce an unexpected value if the value of the unsigned primitive cannot be represented using a signed primitive.
CWE-197	**Numeric Truncation Error**	Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion. When a primitive is cast to a smaller primitive, the high order bits of the large value are lost in the conversion, potentially resulting in an unexpected value that is not equal to the original value. This value may be required as an index into a buffer, a loop iterator, or simply necessary state data. In any case, the value cannot be trusted and the system will be in an undefined state. While this method may be employed viably to isolate the low bits of a value, this usage is rare, and truncation usually implies that an implementation error has occurred.
CWE-682	**Incorrect Calculation**	The software performs a calculation that generates incorrect or unintended results that are later used in security-critical decisions or resource management.
CWE-131	**Incorrect Calculation of Buffer Size**	The software does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.

CWE-369	**Divide By Zero**	The product divides a value by zero.
CWE-732	**Incorrect Permission Assignment for Critical Resource**	The software specifies permissions for a security-critical resource in a way that allows that resource to be read or modified by unintended actors.
CWE-778	**Insufficient Logging**	When a security-critical event occurs, the software either does not record the event or omits important details about the event when logging it.
CWE-783	**Operator Precedence Logic Error**	The program uses an expression in which operator precedence causes incorrect logic to be used. While often just a bug, operator precedence logic errors can have serious consequences if they are used in security-critical code, such as making an authentication decision.
CWE-789	**Uncontrolled Memory Allocation**	The product allocates memory based on an untrusted size value, but it does not validate or incorrectly validates the size, allowing arbitrary amounts of memory to be allocated.
CWE-798	**Use of Hard-coded Credentials**	The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.
CWE-259	**Use of Hard-coded Password**	The software contains a hard-coded password, which it uses for its own inbound authentication or for outbound communication to external components.
CWE-321	**Use of Hard-coded Cryptographic Key**	The use of a hard-coded cryptographic key significantly increases the possibility that encrypted data may be recovered.
CWE-835	**Loop with Unreachable Exit Condition ('Infinite Loop')**	The program contains an iteration or loop with an exit condition that cannot be reached, i.e., an infinite loop.
CWE-917	**Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection')**	The software constructs all or part of an expression language (EL) statement in a Java Server Page (JSP) using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended EL statement before it is executed.

CWE-1057	**Data Access Operations Outside of Expected Data Manager Component**	The software uses a dedicated, central data manager component as required by design, but it contains code that performs data-access operations that do not use this data manager. Notes: • The dedicated data access component can be either client-side or server-side, which means that data access components can be developed using non-SQL language. • If there is no dedicated data access component, every data access is a weakness. • For some embedded software that requires access to data from anywhere, the whole software is defined as a data access component. This condition must be identified as input to the analysis.

3.3 Reliability

Reliability measures of the extent to which software contains weaknesses that cause outages, unexpected behavior, instability, data corruption, long recovery times, or other related problems. The weaknesses that compose the CISQ Automated Source Code Reliability Measure are presented in Table 3. This measure contains 35 parent weaknesses and 39 contributing weaknesses (children in the CWE) that represent variants of these weaknesses. The CWE numbers for contributing weaknesses are presented in light blue cells immediately below the parent weakness whose CWE number is in a dark blue cell.

Table 1. Weaknesses composing the Automated Source Code Reliability Measure

CWE #	Descriptor	Weakness description
CWE-119	**Improper Restriction of Operations within the Bounds of a Memory Buffer**	The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.
CWE-120	**Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')**	The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.
CWE-123	**Write-what-where condition**	Any condition where the attacker has the ability to write an arbitrary value to be written to an arbitrary location, often as the result of a buffer overflow.
CWE-125	**Out-of-bounds read**	The software reads data past the end, or before the beginning, of the intended buffer.
CWE-130	**Improper Handling of Length Parameter Inconsistency**	The software parses a formatted message or structure, but it does not handle or incorrectly handles a length field that is inconsistent with the actual length of the associated data.
CWE-786	**Access of Memory Location Before Start of Buffer**	The software reads or writes to a buffer using an index or pointer that references a memory location prior to the beginning of the buffer. This typically occurs when a pointer or its index is decremented to a position before the buffer, when pointer arithmetic results in a position before the beginning of the valid memory location, or when a negative index is used.
CWE-787	**Out-of-bounds Write**	The software writes data past the end, or before the beginning, of the intended buffer. The software may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer.
CWE-788	**Access of Memory Location After End of Buffer**	The software reads or writes to a buffer using an index or pointer that references a memory location after the end of the buffer. This typically occurs when a pointer or its index is decremented to a position before the buffer; when pointer arithmetic results in a position before the buffer; or when a negative index is used, which generates a position before the buffer.

CWE-805	**Buffer Access with Incorrect Length Value**	The software uses a sequential operation to read or write a buffer, but it uses an incorrect length value that causes it to access memory that is outside of the bounds of the buffer.
CWE-822	**Untrusted Pointer Dereference**	The program obtains a value from an untrusted source, converts this value to a pointer, and dereferences the resulting pointer. There are several variants of this weakness, including but not necessarily limited to: • The untrusted value is directly invoked as a function call. • In OS kernels or drivers where there is a boundary between "userland" and privileged memory spaces, an untrusted pointer might enter through an API or system call (see CWE-781 for one such example). • Inadvertently accepting the value from an untrusted control sphere when it did not have to be accepted as input at all. This might occur when the code was originally developed to be run by a single user in a non-networked environment, and the code is then ported to or otherwise exposed to a networked environment.
CWE-823	**Use of Out-of-range Pointer Offset**	The program performs pointer arithmetic on a valid pointer, but it uses an offset that can point outside of the intended range of valid memory locations for the resulting pointer. • While a pointer can contain a reference to any arbitrary memory location, a program typically only intends to use the pointer to access limited portions of memory, such as contiguous memory used to access an individual array. • Programs may use offsets to access fields or sub-elements stored within structured data. The offset might be out-of-range if it comes from an untrusted source, is the result of an incorrect calculation, or occurs because of another error.
CWE-824	**Access of Uninitialized Pointer**	The program accesses or uses a pointer that has not been initialized. If the pointer contains an uninitialized value, then the value might not point to a valid memory location.
CWE-825	**Expired Pointer Dereference**	The program dereferences a pointer that contains a location for memory that was previously valid, but is no longer valid.
CWE-170	**Improper Null Termination**	The software does not terminate or incorrectly terminates a string or array with a null character or equivalent terminator.
CWE-252	**Unchecked Return Value**	The software does not check the return value from a method or function, which can prevent it from detecting unexpected states and conditions.

CWE-390	**Detection of Error Condition Without Action**	The software detects a specific error, but takes no actions to handle the error. For instance, where an exception handling block (such as Catch and Finally blocks) do not contain any instruction, making it impossible to accurately identify and adequately respond to unusual and unexpected conditions.
CWE-394	**Unexpected Status Code or Return Value**	The software does not properly check when a function or operation returns a value that is legitimate for the function, but is not expected by the software.
CWE-404	**Improper Resource Shutdown or Release**	The program does not release or incorrectly releases a resource before it is made available for re-use.
CWE-401	**Improper Release of Memory Before Removing Last Reference ('Memory Leak')**	The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.
CWE-772	**Missing Release of Resource after Effective Lifetime**	The software does not release a resource after its effective lifetime has ended, i.e., after the resource is no longer needed.
CWE-775	**Missing Release of File Descriptor or Handle after Effective Lifetime**	The software does not release a file descriptor or handle after its effective lifetime has ended, i.e., after the file descriptor/handle is no longer needed. When a file descriptor or handle is not released after use (typically by explicitly closing it), attackers can cause a denial of service by consuming all available file descriptors/handles, or otherwise preventing other system processes from obtaining their own file descriptors/handles.
CWE-424	**Improper Protection of Alternate Path**	The product does not sufficiently protect all possible paths that a user can take to access restricted functionality or resources. When data storage relies on a DBMS, special care shall be given to secure all data accesses and ensure data integrity.
CWE-459	**Incomplete Cleanup**	The software does not properly "clean up" and remove temporary or supporting resources after they have been used.
CWE-476	**NULL Pointer Dereference**	A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.
CWE-480	**Use of Incorrect Operator**	The programmer accidentally uses the wrong operator, which changes the application logic in security-relevant ways.
CWE-484	**Omitted Break Statement in Switch**	The program omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. This can cause problems when the programmer only intended to execute code associated with one condition.

CWE-562	**Return of Stack Variable Address**	A function returns the address of a stack variable, which will cause unintended program behavior, typically in the form of a crash. Because local variables are allocated on the stack, when a program returns a pointer to a local variable, it is returning a stack address. A subsequent function call is likely to re-use this same stack address, thereby overwriting the value of the pointer, which no longer corresponds to the same variable since a function's stack frame is invalidated when it returns. At best this will cause the value of the pointer to change unexpectedly. In many cases it causes the program to crash the next time the pointer is dereferenced.
CWE-595	**Comparison of Object References Instead of Object Contents**	The program compares object references instead of the contents of the objects themselves, preventing it from detecting equivalent objects.
CWE-597	**Use of Wrong Operator in String Comparison**	The software uses the wrong operator when comparing a string, such as using "==" when the equals() method should be used instead. In Java, using == or != to compare two strings for equality actually compares two objects for equality, not their values.
CWE-1097	**Persistent Storable Data Element without Associated Comparison Control Element**	The software uses a storable data element that does not have all of the associated functions or methods that are necessary to support comparison. Remove instances where the persistent data has missing or improper dedicated comparison operations. Note: * In case of technologies with classes, this means situations where a persistent field is from a class that is made persistent while it does not implement methods from the list of required comparison operations (a JAVA example is the list composed of {'hashCode()','equals()'} methods)
CWE-662	**Improper Synchronization**	The software attempts to use a shared resource in an exclusive manner, but does not prevent or incorrectly prevents use of the resource by another thread or process.
CWE-366	**Race Condition within a Thread**	If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.
CWE-543	**Use of Singleton Pattern Without Synchronization in a Multithreaded Context**	The software uses the singleton pattern when creating a resource within a multithreaded environment.
CWE-567	**Unsynchronized Access to Shared Data in a Multithreaded Context**	The product does not properly synchronize shared data, such as static variables across threads, which can lead to undefined behavior and unpredictable data changes.
CWE-667	**Improper Locking**	The software does not properly acquire a lock on a resource, or it does not properly release a lock on a resource, leading to unexpected resource state changes and behaviors.

CWE-764	**Multiple Locks of a Critical Resource**	The software locks a critical resource more times than intended, leading to an unexpected state in the system.
CWE-820	**Missing Synchronization**	The software utilizes a shared resource in a concurrent manner but does not attempt to synchronize access to the resource.
CWE-821	**Incorrect Synchronization**	The software utilizes a shared resource in a concurrent manner but it does not correctly synchronize access to the resource.
CWE-1058	**Invokable Control Element in Multi-Thread Context with non-Final Static Storable or Member Element**	The code contains a function or method that operates in a multi-threaded environment but owns an unsafe non-final static storable or member data element.
CWE-1096	**Singleton Class Instance Creation without Proper Locking or Synchronization**	The software implements a Singleton design pattern but does not use appropriate locking or other synchronization mechanism to ensure that the singleton class is only instantiated once.
CWE-665	**Improper Initialization**	The software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state when it is accessed or used.
CWE-456	**Missing Initialization of a Variable**	The software does not initialize critical variables, which causes the execution environment to use unexpected values.
CWE-457	**Use of uninitialized variable**	The code uses a variable that has not been initialized, leading to unpredictable or unintended results.
CWE-672	**Operation on a Resource after Expiration or Release**	The software uses, accesses, or otherwise operates on a resource after that resource has been expired, released, or revoked.
CWE-415	**Double Free**	The product calls free() twice on the same memory address, potentially leading to modification of unexpected memory locations.
CWE-416	**Use After Free**	Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code.
CWE-681	**Incorrect Conversion between Numeric Types**	When converting from one data type to another, such as long to integer, data can be omitted or translated in a way that produces unexpected values. If the resulting values are used in a sensitive context, then dangerous behaviors may occur. For instance, if the software declares a variable, field, member, etc. with a numeric type, and then updates it with a value from a second numeric type that is incompatible with the first numeric type.

CWE-194	**Unexpected Sign Extension**	The software performs an operation on a number that causes it to be sign-extended when it is transformed into a larger data type. When the original number is negative, this can produce unexpected values that lead to resultant weaknesses.
CWE-195	**Signed to Unsigned Conversion Error**	The software uses a signed primitive and performs a cast to an unsigned primitive, which can produce an unexpected value if the value of the signed primitive cannot be represented using an unsigned primitive.
CWE-196	**Unsigned to Signed Conversion Error**	The software uses an unsigned primitive and performs a cast to a signed primitive, which can produce an unexpected value if the value of the unsigned primitive cannot be represented using a signed primitive.
CWE-197	**Numeric Truncation Error**	Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion. When a primitive is cast to a smaller primitive, the high order bits of the large value are lost in the conversion, potentially resulting in an unexpected value that is not equal to the original value. This value may be required as an index into a buffer, a loop iterator, or simply necessary state data. In any case, the value cannot be trusted and the system will be in an undefined state. While this method may be employed viably to isolate the low bits of a value, this usage is rare, and truncation usually implies that an implementation error has occurred.
CWE-682	**Incorrect Calculation**	The software performs a calculation that generates incorrect or unintended results that are later used in security-critical decisions or resource management.
CWE-131	**Incorrect Calculation of Buffer Size**	The software does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.
CWE-369	**Divide By Zero**	The product divides a value by zero.
CWE-703	**Improper Check or Handling of Exceptional Conditions**	The software does not properly anticipate or handle exceptional conditions that rarely occur during normal operation of the software.
CWE-248	**Uncaught Exception**	An exception is thrown from a function, but it is not caught.
CWE-391	**Unchecked Error Condition**	Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.
CWE-392	**Missing Report of Error Condition**	The software encounters an error but does not provide a status code or return value to indicate that an error has occurred.
CWE-704	**Incorrect Type Conversion or Cast**	The software does not correctly convert an object, resource, or structure from one type to a different type.

CWE-758	**Reliance on Undefined, Unspecified, or Implementation-Defined Behavior**	The software uses an API function, data structure, or other entity in a way that relies on properties that are not always guaranteed to hold for that entity.
CWE-833	**Deadlock**	The software contains multiple threads or executable segments that are waiting for each other to release a necessary lock, resulting in deadlock.
CWE-835	**Loop with Unreachable Exit Condition ('Infinite Loop')**	The program contains an iteration or loop with an exit condition that cannot be reached, i.e., an infinite loop.
CWE-908	**Use of Uninitialized Resource**	The software uses a resource that has not been properly initialized.
CWE-1045	**Parent Class with a Virtual Destructor and a Child Class without a Virtual Destructor**	A parent class has a virtual destructor method, but the parent has a child class that does not have a virtual destructor.
CWE-1051	**Initialization with Hard-Coded Network Resource Configuration Data**	The software initializes data using hard-coded values that act as as network resource identifiers.
CWE-1066	**Missing Serialization Control Element**	The software contains a serializable data element that does not have an associated serialization method.
CWE-1070	**Serializable Data Element Containing non-Serializable Item Elements**	The software contains a serializable, storable data element such as a field or member, but the data element contains member elements that are not serializable.
CWE-1077	**Floating Point Comparison with Incorrect Operator**	The code performs a comparison such as an equality test between two float (floating point) values, but it uses comparison operators that do not account for the possibility of loss of precision. Numeric calculation using floating point values can generate imprecise results because of rounding errors. As a result, two different calculations might generate numbers that are mathematically equal, but have slightly different bit representations that do not translate to the same mathematically-equal values. As a result, an equality test or other comparison might produce unexpected results. (an example in JAVA, is the use of '= =' or '! =') instead of being checked for precision.
CWE-1079	**Parent Class without Virtual Destructor Method**	A parent class contains one or more child classes, but the parent class does not have a virtual destructor method.

CWE-1082	**Class Instance Self Destruction Control Element**	The code contains a class instance that calls the method or function to delete or destroy itself. (an example of a self-destruction in C++ is 'delete this')
CWE-1083	**Data Access from Outside Designated Data Manager Component**	The software is intended to manage data access through a particular data manager component such as a relational or non-SQL database, but it contains code that performs data access operations without using that component. Notes: • The dedicated data access component can be either client-side or server-side, which means that data access components can be developed using non-SQL language. • If there is no dedicated data access component, every data access is a violation. • For some embedded software that requires access to data from anywhere, the whole software is defined as a data access component. This condition must be identified as input to the analysis.
CWE-1087	**Class with Virtual Method without a Virtual Destructor**	A class contains a virtual method, but the method does not have an associated virtual destructor.
CWE-1088	**Synchronous Access of Remote Resource without Timeout**	The code has a synchronous call to a remote resource, but there is no timeout for the call, or the timeout is set to infinite.
CWE-1098	**Data Element containing Pointer Item without Proper Copy Control Element**	The code contains a data element with a pointer that does not have an associated copy or constructor method.

3.4 Performance Efficiency

Performance Efficiency measures the extent to which software contains weaknesses that can degrade a system's performance or cause excessive use of processor, memory, or other resources. The weaknesses that compose the CISQ Automated Source Code Performance Efficiency Measure are presented in Table 4. This measure contains 16 parent weaknesses and 3 contributing weaknesses (children in the CWE) that represent variants of these weaknesses. The CWE numbers for contributing weaknesses are presented in light blue cells immediately below the parent weakness whose CWE number is in a dark blue cell.

Table 4. Weaknesses Composing Automated Source Code Performance Efficiency Measure

CWE #	Descriptor	Weakness Description
CWE-404	**Improper Resource Shutdown or Release**	The program does not release or incorrectly releases a resource before it is made available for re-use.
CWE-401	**Improper Release of Memory Before Removing Last Reference ('Memory Leak')**	The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.
CWE-772	**Missing Release of Resource after Effective Lifetime**	The software does not release a resource after its effective lifetime has ended, i.e., after the resource is no longer needed.
CWE-775	**Missing Release of File Descriptor or Handle after Effective Lifetime**	The software does not release a file descriptor or handle after its effective lifetime has ended, i.e., after the file descriptor/handle is no longer needed. When a file descriptor or handle is not released after use (typically by explicitly closing it), attackers can cause a denial of service by consuming all available file descriptors/handles, or otherwise preventing other system processes from obtaining their own file descriptors/handles.
CWE-424	**Improper Protection of Alternate Path**	The product does not sufficiently protect all possible paths that a user can take to access restricted functionality or resources. When data storage relies on a DBMS, special care shall be given to secure all data accesses and ensure data integrity.
CWE-1042	**Static Member Data Element outside of a Singleton Class Element**	The code contains a member element that is declared as static (but not final), in which its parent class element is not a singleton class - that is, a class element that can be used only once in the 'to' association of a Create action.
CWE-1043	**Data Element Aggregating an Excessively Large Number of Non-Primitive Elements**	The software uses a data element that has an excessively large number of sub-elements with non-primitive data types such as structures or aggregated objects. (default threshold for the maximum number of aggregated non-primitive data types is 5, *alternate threshold can be set prior to analysis*).

CWE-1046	**Creation of Immutable Text Using String Concatenation**	This programming pattern can be inefficient in comparison with use of text buffer data elements. This issue can make the software perform more slowly. If the relevant code is reachable by an attacker, then this performance problem might introduce a vulnerability.
CWE-1049	**Excessive Data Query Operations in a Large Data Table**	The software performs a data query with a large number of joins and sub-queries on a large data table. (default thresholds are 5 joins, 3 sub-queries, and 1,000,000 rows for a large table, *alternate thresholds for all three parameters can be set prior to analysis*).
CWE-1050	**Excessive Platform Resource Consumption within a Loop**	The software has a loop body or loop condition that contains a control element that directly or indirectly consumes platform resources, e.g. messaging, sessions, locks, or file descriptors. (default threshold for resource consumption should be set based on the system architecture *prior to analysis*).
CWE-1057	**Data Access Operations Outside of Expected Data Manager Component**	The software uses a dedicated, central data manager component as required by design, but it contains code that performs data-access operations that do not use this data manager. Notes: · The dedicated data access component can be either client-side or server-side, which means that data access components can be developed using non-SQL language. · If there is no dedicated data access component, every data access is a weakness. · For some embedded software that requires access to data from anywhere, the whole software is defined as a data access component. This condition must be identified as input to the analysis.
CWE-1060	**Excessive Number of Inefficient Server-Side Data Accesses**	The software performs too many data queries without using efficient data processing functionality such as stored procedures. (default threshold for maximum number of data queries is **5**, *alternate threshold can be set prior to analysis*).
CWE-1067	**Excessive Execution of Sequential Searches of Data Resource**	The software contains a data query against a SQL table or view that is configured in a way that does not utilize an index and may cause sequential searches to be performed. (default threshold for a weakness to be counted is a query on a table of at least 500 rows, or an alternate threshold recommended by the database vendor. No weakness should be counted under conditions where the vendor recommends an index should not be used. An *alternate threshold can be set prior to analysis*).

CWE-1072	**Data Resource Access without Use of Connection Pooling**	The software accesses a data resource through a database without using a connection pooling capability. (the use of a connection pool is technology dependent; for example, connection pooling is disabled with the addition of 'Pooling=false' to the connection string with ADO.NET or the value of a 'com.sun.jndi.ldap.connect.pool' environment parameter in Java).
CWE-1073	**Non-SQL Invokable Control Element with Excessive Number of Data Resource Accesses**	The software contains a client with a function or method that contains a large number of data accesses/queries that are sent through a data manager, i.e., does not use efficient database capabilities. (default threshold for the maximum number of data queries is 2, *alternate threshold can be set prior to analysis*).
CWE-1089	**Large Data Table with Excessive Number of Indices**	The software uses a large data table (default is 1,000,000 rows; *alternate threshold can be set prior to analysis*) that contains an excessively large number of indices. (default threshold for the maximum number of indices is 3, *alternate threshold can be set prior to analysis*).
CWE-1091	**Use of Object without Invoking Destructor Method**	The software contains a method that accesses an object but does not later invoke the element's associated finalize/destructor method.
CWE-1094	**Excessive Index Range Scan for a Data Resource**	The software contains an index range scan for a large data table, (default threshold is 1,000,000 rows, *alternate threshold can be set prior to analysis*) but the scan can cover a large number of rows. (default threshold for the index range is 10, *alternate threshold can be set prior to analysis).*

3.5 Maintainability

Maintainability measures the extent to which software contains weaknesses that make software hard to understand or change, resulting in excessive maintenance time and cost as well as higher defect injection rates. The quality measure elements (weaknesses violating software quality rules) that compose the CISQ Automated Source Code Maintainability Measure are presented in Table 5. This measure contains 29 parent weaknesses and no contributing weaknesses.

Table 5. Quality Measure Elements for Automated Source Code Maintainability Measure

CWE #	Descriptor	Weakness Description
CWE-407	Algorithmic Complexity	An algorithm in a product has an inefficient worst-case computational complexity that may be detrimental to system performance and can be triggered by an attacker, typically using crafted manipulations that ensure that the worst case is being reached.
CWE-478	Missing Default Case in Switch Statement	The code does not have a default case in a switch statement, which might lead to complex logical errors and resultant weaknesses.
CWE-480	Use of Incorrect Operator	The programmer accidentally uses the wrong operator, which changes the application logic in security-relevant ways.
CWE-484	Omitted Break Statement in Switch	The program omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. This can cause problems when the programmer only intended to execute code associated with one condition.
CWE-561	Dead code	The software contains dead code that can never be executed. (Thresholds are set at 5% logically dead code or 0% for code that is structurally dead. Code that exists in the source but not in the object does not count.)
CWE-570	Expression is Always False	The software contains an expression that will always evaluate to false.
CWE-571	Expression is Always True	The software contains an expression that will always evaluate to true.
CWE-783	Operator Precedence Logic Error	The program uses an expression in which operator precedence causes incorrect logic to be used.
CWE-1041	Use of Redundant Code (Copy-Paste)	The software has multiple functions, methods, procedures, macros, etc. that contain the same code. (The default threshold for each instance of copy-pasted code sets the maximum number of allowable copy-pasted instructions at 10% of the total instructions in the instance, *alternate thresholds can be set prior to analysis*).

CWE-1045	**Parent Class with a Virtual Destructor and a Child Class without a Virtual Destructor**	A parent class has a virtual destructor method, but the parent has a child class that does not have a virtual destructor.
CWE-1047	**Modules with Circular Dependencies**	The software contains modules in which one module has references that cycle back to itself, i.e., there are circular dependencies.
CWE-1048	**Invokable Control Element with Large Number of Outward Calls (Excessive Coupling or Fan-out)**	The code contains callable control elements that contain an excessively large number of references to other application objects external to the context of the callable, i.e. a Fan-Out value that is excessively large. (default threshold for the maximum number of references is 5, *alternate threshold can be set prior to analysis*)
CWE-1051	**Initialization with Hard-Coded Network Resource Configuration Data**	The software initializes data using hard-coded values that act as network resource identifiers.
CWE-1052	**Excessive Use of Hard-Coded Literals in Initialization**	The software initializes a data element using a hard-coded literal that is not a simple integer or static constant element.
CWE-1054	**Invocation of a Control Element at an Unnecessarily Deep Horizontal Layer (Layer-skipping Call)**	The code at one architectural layer invokes code that resides at a deeper layer than the adjacent layer, i.e., the invocation skips at least one layer, and the invoked code is not part of a vertical utility layer that can be referenced from any horizontal layer.
CWE-1055	**Multiple Inheritance from Concrete Classes**	The software contains a class with inheritance from more than one concrete class.
CWE-1062	**Parent Class Element with References to Child Class**	The code has a parent class that contains references to a child class, its methods, or its members.
CWE-1064	**Invokable Control Element with Signature Containing an Excessive Number of Parameters**	The software contains a function, subroutine, or method whose signature has an unnecessarily large number of parameters/arguments. (default threshold for the maximum number of parameters is 7, *alternate threshold can be set prior to analysis*).
CWE-1074	**Class with Excessively Deep Inheritance**	A class has an inheritance level that is too high, i.e., it has a large number of parent classes. (default threshold for maximum Inheritance levels is 7, *alternate threshold can be set prior to analysis*).
CWE-1075	**Unconditional Control Flow Transfer outside of Switch Block**	The software performs unconditional control transfer (such as a "goto") in code outside of a branching structure such as a switch block.
CWE-1079	**Parent Class without Virtual Destructor Method**	A parent class contains one or more child classes, but the parent class does not have a virtual destructor method.

CWE-1080	**Source Code File with Excessive Number of Lines of Code**	A source code file has too many lines of code. (default threshold for the maximum lines of code is 1000, *alternate threshold can be set prior to analysis*).
CWE-1084	**Invokable Control Element with Excessive File or Data Access Operations**	A function or method contains too many operations that utilize a data manager or file resource. (default threshold for the maximum number of SQL or file operations is 7, *alternate threshold can be set prior to analysis*).
CWE-1085	**Invokable Control Element with Excessive Volume of Commented-out Code**	A function, method, procedure, etc. contains an excessive amount of code that has been commented out within its body. (default threshold for the maximum percent of commented-out instructions is 2%, *alternate threshold can be set prior to analysis*).
CWE-1086	**Class with Excessive Number of Child Classes**	A class contains an unnecessarily large number of children. (default threshold for the maximum number of children of a class is 10, *alternate threshold can be set prior to analysis*).
CWE-1087	**Class with Virtual Method without a Virtual Destructor**	A class contains a virtual method, but the method does not have an associated virtual destructor.
CWE-1090	**Method Containing Access of a Member Element from Another Class**	A method for a class performs an operation that directly accesses a member element from another class.
CWE-1095	**Loop Condition Value Update within the Loop**	The software uses a loop with a control flow condition based on a value that is updated within the body of the loop.
CWE-1121	**Excessive McCabe Cyclomatic Complexity**	A module, function, method, procedure, etc. contains McCabe cyclomatic complexity that exceeds a desirable maximum. (default threshold for Cyclomatic Complexity is 20, *alternate threshold can be set prior to analysis*).

Appendix A: Certification Measurement Reports

Software Quality Certification

Based on CISQ Automated Source Code Quality Measures

This report describes the analysis results for <<application>> <<release>> that was analyzed and measured on <<date>> by <<service provider>> using the <<tool>> developed by <<vendor>>. This analysis certifies the level of quality measured in this application when measured against the CISQ Quality Characteristic Measures developed by the Consortium for IT Software Quality and adopted as standards by the Object Management Group (OMG), an international IT standards organization. These measures are developed from counting the number of times critical rules of good architectural and coding practice for each characteristic have been violated. Since structural quality analysis tools differ in the violations of good architectural and coding practices they can detect, the tables will only include results for practices that were evaluated and are the basis for this certification. For each architectural or coding practice within each quality characteristic, the tables below present both the number of times each practice was violated and the number of opportunities for the practice to have been violated within the application. When aggregated over the all violations, these numbers provide the basis for a 6-sigma ranking for each quality characteristic and the aggregated characteristics. That is, the σ level representing the number of violations per million opportunities. This certification provides an evidence-based assessment of the risk this application poses to the business operations it supports or its cost of ownership.

A.1 Example Summary Results

Quality Characteristics	Weaknesses	Opportunities	6-sigma level
Total Application Quality			
Reliability			
Security			
Performance Efficiency			
Maintainability			

A.2 Security Example

ASCSM #	Architectural or Coding Practice	Vios	Opps	6σ
CWE-22	Avoid failure to sanitize user input in path manipulation operations			
CWE-78	Avoid failure to sanitize user input used as operating system commands			
CWE-79	Avoid failure to sanitize user input used in output generation operations (cross-site scripting)			
CWE-89	Avoid failure to sanitize user input used in SQL compilation operations			
CWE-99	Avoid failure to sanitize user input in use as resource names or references			
CWE-120	Avoid buffer operations among buffers with incompatible sizes			
CWE-129	Avoid failure to check range of user input used as array index			
CWE-134	Avoid failure to sanitize user input used in formatting operations			
CWE-252-resource	Avoid improper processing of the execution status of resource handling operations			
CWE-327	Avoid failure to use vetted cryptographic libraries			
CWE-396	Avoid catch units that catch overly broad exception data types			
CWE-397	Avoid throwing overly broad exception data types			
CWE-434	Avoid failure to sanitize user input used in file upload operations			
CWE-456	Avoid failure to explicitly initialize software data elements in use			
CWE-606	Avoid failure to check range of user input used in iteration control			
CWE-667	Avoid improper locking of shared data			
CWE-672	Avoid access to a released, revoked, or expired resource			
CWE-681	Avoid conversions between incompatible numerical data types			
CWE-772	Avoid failure to release resources after their usage lifetime ends			
CWE-789	Avoid failure to relase a platform resource after its useful lifetime			
CWE-798	Avoid using hard-coded credentials for remote authentication			
CWE-835	Avoid infinite loops resulting from unreachable exit conditions			
Total aggregated Security violation results				

Appendix B: Java-EE Weakness Pattern Examples

These tables present examples of weakness patterns in Java-EE to provide a guideance on the patterns a conformant technology should be able to detect. If a cell is empty, then it is assumed there are no structures unique to Java EE that are required beyond the general capability for detecting the elements of the pattern. An 'N/A' indicates the weakness cannot occur in Java EE.

B.1 Reliability Examples

CISQ identifier	JEE detection aspects	JEE-specific example

CWE-120	• mostly N/A,
	• except for JNI (JAVA Native Interface) and for system calls
		• OWASP unsafe JNI
		
		class Echo {
		public native void runEcho();
		static {
		
		System.loadLibrary("echo");
		}
		public static void main(String[] args) {
		new Echo().runEcho();
		}
		}
		with
		#include <jni.h>
		#include "Echo.h"//the java class above compiled with javah
		#include <stdio.h>
		
		JNIEXPORT void JNICALL
		Java_Echo_runEcho(JNIEnv *env, jobject obj)
		{
		char buf[64];
		gets(buf);
		printf(buf);
		}

CWE-252-data	• The traditional defense of this coding error is: "I know the requested value will always exist because.... If it does not exist, the program cannot perform the desired behavior so it doesn't matter whether I handle the error or simply allow the program to die dereferencing a null value." But attackers are skilled at finding unexpected paths through programs, particularly when exceptions are involved. (Src. CWE-252)	

CWE-252-resource	• The traditional defense of this coding error is: "I know the requested value will always exist because.... If it does not exist, the program cannot perform the desired behavior so it doesn't matter whether I handle the error or simply allow the program to die dereferencing a null value." But attackers are skilled at finding unexpected paths through programs, particularly when exceptions are involved. (Src. CWE-252)	• CWE-252 sample
		•
		FileInputStream fis;
		
		byte[] byteArray = new byte[1024];
		for (Iterator i=users.iterator(); i.hasNext();) {
		String userName = (String) i.next();
		String pFileName = PFILE_ROOT + "/" + userName;
		FileInputStream fis = new FileInputStream(pFileName);
		fis.read(byteArray); // the file is always 1k bytes
		fis.close();
		processPFile(userName, byteArray);
		The code loops through a set of users, reading a private data file for each user. The programmer assumes that the files are always 1 kilobyte in size and therefore ignores the return value from Read(). If an attacker can create a smaller file, the program will recycle the remainder of the data from the previous user and treat it as though it belongs to the attacker.

CWE-396	• (none so far)	• QR-7962 try { ,,, } catch (Exception /*e*/) // <= VIOLATION { ,,, }

CWE-397	• (none so far)	• QR-7824
		
		void f()
		{
		,,,
		throw new Exception("My Message"); // <= VIOLATION
		,,,
		}

CWE-456	• (none so far)
	• not N/A (cf. samples)
		• CWE-456 samples
		• private User user;
		• public void someMethod() {
		• // Do something interesting.
		• ...
		• // Throws NPE if user hasn't been properly initialized.
		• String username = user.getName();
		}
		public class BankManager {
		// user allowed to perform bank manager tasks
		private User user = null;
		private boolean isUserAuthentic = false;
		// constructor for BankManager class
		public BankManager() {
		...
		}
		// retrieve user from database of users
		public User getUserFromUserDatabase(String username){
		...
		}
		// set user variable using username
		public void setUser(String username) {
		this.user = getUserFromUserDatabase(username);
		}
		// authenticate user
		public boolean authenticateUser(String username, String password) {
		if (username.equals(user.getUsername()) && password.equals(user.getPassword())) {
		isUserAuthentic = true;
		}
		return isUserAuthentic;
		}
		// methods for performing bank manager tasks
		...
		}

CWE-674	• (none so far)	
CWE-704	• (none so far)	

CWE-772	• Check resource types of the pattern search: ○ stream ○ file ○ lock ○ thread (not so simple as the thread itself must be wired to cooperate so the pattern is not enough) ○ ... • Check usage of data flow link	
CWE-788	• N/A	

Appendix C: CISQ

The purpose of the Consortium for IT Software Quality (CISQ) is to develop specifications for automated measures of software quality characteristics taken on source code. These measures were designed to provide international standards for measuring software structural quality that can be used by IT organizations, IT service providers, and software vendors in contracting, developing, testing, accepting, and deploying IT software applications. Executives from the member companies that joined CISQ prioritized the quality characteristics of Reliability, Security, Performance Efficiency, and Maintainability to be developed as measurement specifications.

CISQ strives to maintain consistency with ISO/IEC standards to the extent possible, and in particular with the ISO/IEC 25000 series that replaces ISO/IEC 9126 and defines quality measures for software systems. In order to maintain consistency with the quality model presented in ISO/IEC 25010, software quality characteristics are defined for the purpose of this specification as attributes that can be measured from the static properties of software, and can be related to the dynamic properties of a computer system as affected by its software. However, the 25000 series, and in particular ISO/IEC 25023 which elaborates quality characteristic measures, does not define these measures at the source code level. Thus, this and other CISQ quality characteristic specifications supplement ISO/IEC 25023 by providing a deeper level of software measurement, one that is rooted in measuring software attributes in the source code.

Companies interested in joining CISQ held executive forums in Frankfurt, Germany; Arlington, VA; and Bangalore, India to set strategy and direction for the consortium. In these forums four quality characteristics were selected as the most important targets for automation—reliability, security, performance efficiency, and maintainability. These attributes cover four of the eight quality characteristics described in ISO/IEC 25010. Figure 1 displays the ISO/IEC 25010 software product quality model with the four software quality characteristics selected for automation by CISQ highlighted in orange. Each software quality characteristic is shown with the sub-characteristics that compose it.