**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc174309290)

[Contents 1](#_Toc174309291)

[Overview 2](#_Toc174309292)

[Purpose 2](#_Toc174309293)

[Scope 2](#_Toc174309294)

[Module Three Milestone 2](#_Toc174309295)

[Ten Core Security Principles 2](#_Toc174309296)

[C/C++ Ten Coding Standards 3](#_Toc174309297)

[Coding Standard 1 4](#_Toc174309298)

[Coding Standard 2 6](#_Toc174309299)

[Coding Standard 3 9](#_Toc174309300)

[Coding Standard 4 11](#_Toc174309301)

[Coding Standard 5 14](#_Toc174309302)

[Coding Standard 6 17](#_Toc174309303)

[Coding Standard 7 20](#_Toc174309304)

[Coding Standard 8 23](#_Toc174309305)

[Coding Standard 9 27](#_Toc174309306)

[Coding Standard 10 31](#_Toc174309307)

[Defense-in-Depth Illustration 35](#_Toc174309308)

[Automation 35](#_Toc174309309)

[Summary of Risk Assessments 36](#_Toc174309310)

[Policies for Encryption and Triple A 36](#_Toc174309311)

[Audit Controls and Management 38](#_Toc174309312)

[Enforcement 38](#_Toc174309313)

[Exceptions Process 38](#_Toc174309314)

[Distribution 39](#_Toc174309315)

[Policy Change Control 39](#_Toc174309316)

[Policy Version History 39](#_Toc174309317)

[Appendix A Lookups 39](#_Toc174309318)

[Approved C/C++ Language Acronyms 39](#_Toc174309319)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input validation ensures that data from users or other systems meets the expected format, type, and range before processing. This principle helps prevent security vulnerabilities like SQL injection, cross-site scripting (XSS), and buffer overflows by rejecting or sanitizing invalid or malicious data. By validating inputs, systems can enforce strict data integrity and avoid the execution of harmful inputs that could compromise security. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential issues in Code that could lead to vulnerabilities or unexpected behaviors. By addressing these warnings, developers can identify and resolve subtle problems that might not be apparent during regular testing. Ignoring compiler warnings can lead to security flaws and unstable software, so reviewing and resolving these warnings is essential to maintaining a robust and secure codebase. |
| 1. Architect and Design for Security Policies | Security policies should be integral to the software design and architecture process. This principle involves incorporating security requirements early in the development cycle, ensuring the system is designed with security considerations. By defining and implementing security policies from the outset, developers can build systems that are resilient to threats and easier to maintain securely. |
| 1. Keep It Simple | Complexity often introduces security risks and increases the likelihood of vulnerabilities—the principle of simplicity advocates for designing systems and writing Code that is straightforward to understand. Simple designs reduce the potential attack surface and simplify identifying and fixing security issues. This principle also promotes maintainability and reduces the chances of introducing new security flaws during future modifications. |
| 1. Default Deny | The default deny principle states that all access should be denied by default unless explicitly allowed. This principle minimizes the risk of unauthorized access by ensuring that permissions are granted only to users or systems that have been explicitly authorized. Implementing this principle helps enforce strict access controls and reduces the likelihood of accidental or malicious access to sensitive resources. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege states that users, systems, and processes should only have the minimum level of access necessary to perform their tasks. By limiting privileges, potential security breaches are minimized, and the risk of unauthorized actions or data exposure is reduced. This principle helps confine the scope of access and mitigate the consequences of compromised accounts or components. |
| 1. Sanitize Data Sent to Other Systems | When data is sent from one system to another, it must be sanitized to ensure it does not contain malicious or inappropriate content. This principle involves cleaning and validating data before it is transmitted or processed by other systems. Proper sanitization helps prevent injection attacks and ensures the receiving systems handle data securely and appropriately. |
| 1. Practice Defense in Depth | Defense in depth is a security strategy that involves implementing multiple layers of security controls to protect systems and data. Multiple layers of protection, such as firewalls, encryption, and access controls, strengthen the system's security. This principle ensures that if one layer of defense fails, others are still in place to provide protection, reducing the likelihood of a successful attack. |
| 1. Use Effective Quality Assurance Techniques | Adequate quality assurance (QA) techniques involve rigorous testing and validation of software to identify and fix defects before deployment. QA practices, such as code reviews, static analysis, and dynamic testing, help ensure the software meets security standards and functions as intended. By investing in thorough QA processes, developers can enhance software reliability and security, reducing the likelihood of vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves following established guidelines and best practices for writing secure Code. Secure coding standards provide developers with rules and recommendations to avoid common vulnerabilities and ensure that Code is developed with security in mind. By adhering to these standards, developers can produce Code that is less prone to security issues and easier to maintain securely. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Do not cast to an out-of-range enumeration value: Casting to an out-of-range enumeration value can lead to undefined behavior or unexpected results, as the enumeration may not define valid values for the casted value. Ensuring that the cast is within the defined range of enumeration values helps maintain code reliability and prevents potential errors. |

| **Noncompliant Code** |
| --- |
| This Code casts an integer value (10) to an enum type that only has three defined values (RED, GREEN, BLUE). This results in an out-of-range value that is not defined in the enumeration, leading to undefined behavior. |
| enum Color { RED, GREEN, BLUE }; Color c = static\_cast<Color>(10); *// Casting to an out-of-range value* |

| **Compliant Code** |
| --- |
| Ensure that the value being cast is within the valid range of the enumeration. |
| enum Color { RED, GREEN, BLUE }; Color c = RED; *// Correctly within the range of the enumeration* |

| Principle 2 - Heed Compiler Warnings:  This principle emphasizes the importance of addressing compiler warnings that can indicate potential issues in the Code. Casting to out-of-range enumeration values can result in compiler warnings or errors. By heeding these warnings and ensuring that enumeration values are within the valid range, developers address potential issues before they become runtime problems, thereby preventing undefined behavior and ensuring code reliability. |
| --- |
| Principle 10 - Adopt a Secure Coding Standard:  This principle involves following established guidelines to write secure and reliable Code. The standard of not casting to out-of-range enumeration values aligns with secure coding practices that aim to prevent undefined behavior and errors. By adhering to this standard, developers ensure that their Code is less prone to vulnerabilities and behaves predictably, contributing to overall software security and stability. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | **Low** | **2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **cast-integer-to-enum** | Astrée's cast-integer-to-enum checker identifies unsafe casting operations that convert integer values to enumeration types, which might lead to undefined behavior if the integer is out of the valid range of the enumeration. This helps ensure such operations are correctly handled within the defined bounds of enumeration values. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **C++3013** | The C++3013 checker in Helix QAC focuses on detecting improper casting operations, such as casting to out-of-range enumeration values. This tool helps enforce type safety and ensures cast operations do not lead to undefined or unintended behavior. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **cast-integer-to-enum** | RuleChecker's cast-integer-to-enum checker identifies cases where integers are cast to enumerations, mainly focusing on whether the integer values fall within the valid range of the enumeration. This prevents out-of-range values that could lead to unexpected results. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: INT50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcint50cpp.html) | Polyspace Bug Finder's CERT C++: INT50-CPP checker helps detect unsafe casting operations, ensuring that integer values cast to enumerations are valid. This tool supports maintaining code correctness by preventing potential runtime errors related to casting issues. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not depend on the order of evaluation for side effects: Relying on the order of evaluation of expressions can lead to unpredictable and undefined behavior. Side effects in expressions can vary based on compiler implementation and optimization, so relying on specific evaluation orders can introduce bugs that are hard to trace. |

| **Noncompliant Code** |
| --- |
| The Code relies on the order in which a++ and ++b are evaluated. Different compilers or optimization settings may produce different results, making the Code unreliable. |
| int a = 1; int b = 2; int result = a++ + ++b; *// Dependent on order of evaluation* |

| **Compliant Code** |
| --- |
| Use separate statements to ensure clear and predictable evaluation. |
| int a = 1; int b = 2; a++; b++; int result = a + b; *// Order of evaluation is clear and predictable* |

| Principle 2 - Heed Compiler Warnings:  This principle emphasizes the importance of addressing compiler warnings that can signal potential issues in the Code, such as unpredictable behavior caused by relying on evaluation order. Compilers often warn when expressions with side effects may be evaluated in an unspecified order, which could lead to unintended consequences. By heeding these warnings and ensuring that the Code does not depend on the order of evaluation, developers can avoid subtle bugs and maintain predictable behavior. |
| --- |
| Principle 4 - Keep It Simple:  This principle advocates for simplicity in design and coding to reduce complexity and potential security risks. Dependence on the order of evaluation adds complexity and unpredictability to code. By avoiding reliance on evaluation order, Code remains straightforward and easier to understand, which simplifies debugging and reduces the likelihood of introducing subtle errors. |
| Principle 9 - Use Effective Quality Assurance Techniques:  This principle underscores the need for rigorous testing and validation to ensure software quality. Avoiding dependence on evaluation order is crucial for maintaining code correctness and predictability. Effective quality assurance techniques, such as thorough testing, can help identify issues related to evaluation orders and ensure that Code behaves as expected. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | **Medium** | **3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **evaluation-order**  **multiple-volatile-accesses** | Astrée's evaluation-order and multiple-volatile-accesses checkers identify dependencies on the order of evaluation and issues with accessing volatile variables. They ensure that side effects in expressions are not reliant on evaluation order, which can prevent undefined and unpredictable behavior. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.SE.DEC** **LANG.STRUCT.SE.INC** **LANG.STRUCT.SE.INIT** | CodeSonar's LANG.STRUCT.SE.DEC, LANG.STRUCT.SE.INC, and LANG.STRUCT.SE.INIT checkers detect issues related to side effects in expressions and initialization. They help ensure that Code does not rely on evaluation order, thus avoiding unpredictable behavior. |
| [Splint](https://wiki.sei.cmu.edu/confluence/display/c/Splint) | 3.1.1 |  | Splint provides static analysis to detect various coding issues related to evaluation order and side effects. It helps identify Code that may rely on specific evaluation sequences, thus improving code reliability. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151949) | 3.11 | [IncAndDecMixedWithOtherOperators](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-881) | The SonarQube C/C++ Plugin's IncAndDecMixedWithOtherOperators checker identifies cases where increment and decrement operators are mixed with other operators in a way that may depend on the evaluation order. It helps ensure that Code behaves consistently and avoids undefined behavior. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CLG | Do not attempt to modify string literals: String literals are typically stored in read-only memory. Attempting to modify them results in undefined behavior, which can cause crashes or data corruption. Ensuring string literals remain unmodified protects the program's stability and correctness. |

| **Noncompliant Code** |
| --- |
| This Code attempts to modify a string literal, which is typically stored in a read-only section of memory, leading to undefined behavior. |
| char \*str = "Hello, World!"; str[0] = 'h'; *// Attempting to modify a string literal* |

| **Compliant Code** |
| --- |
| Use writable buffers for modifying strings. |
| char str[] = "Hello, World!"; str[0] = 'h'; *// Modification is safe as str is a writable buffer* |

| Principle 4 - Keep It Simple:  This principle advocates for simplicity in Code to minimize complexity and reduce the likelihood of errors. Modifying string literals can introduce complexity and undefined behavior, leading to instability. By adhering to the standard of not modifying string literals, developers ensure that Code remains straightforward and predictable, reducing the risk of crashes and data corruption. |
| --- |
| Principle 9 - Use Effective Quality Assurance Techniques:  Effective quality assurance involves rigorous testing and validation to catch defects early. Not modifying string literals helps avoid undefined behavior and potential crashes, which should be validated through testing. By following this standard, developers make it easier to test and ensure that string handling is consistent and reliable. |
| Principle 10 - Adopt a Secure Coding Standard:  This principle involves following established guidelines to write secure and reliable Code. The standard of not modifying string literals aligns with secure coding practices that aim to prevent undefined behavior and crashes. Adhering to this standard helps ensure that Code is less prone to vulnerabilities and maintains stability, contributing to overall software security. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | **Medium** | **3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **string-literal-modfication** **write-to-string-literal** | Astrée's string-literal-modfication and write-to-string-literal checkers detect attempts to modify string literals. They help ensure that string literals remain read-only, thus preventing crashes and undefined behavior associated with modifying these literals. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.2 | **C0556, C0752, C0753, C0754**  **C++3063, C++3064, C++3605, C++3606, C++3607** | Helix QAC's checkers C0556, C0752 through C0754, and C++3063 through C++3607 identify cases where string literals might be modified or misused. They ensure that literals remain constant and prevent undefined behavior. |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **157 S** | LDRA Tool Suite's 157 S checker detects modification attempts on string literals. It ensures that literals are used correctly and remain read-only, preventing crashes and undefined behavior. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR30-a** **CERT\_C-STR30-b** | Parasoft C/C++test's CERT\_C-STR30 checkers focus on detecting modifications to string literals. They ensure that literals are not altered, thus maintaining code correctness and avoiding potential instability. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL injection: SQL injection vulnerabilities can occur when user input is improperly handled, leading to unauthorized access or modification of the database. Using parameterized queries or prepared statements prevents such vulnerabilities by separating data from SQL commands. |

| **Noncompliant Code** |
| --- |
| This Code directly includes user input in the SQL query, which is vulnerable to SQL injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';"; |

| **Compliant Code** |
| --- |
| Use parameterized queries to prevent SQL injection. |
| *// Using a parameterized query with a prepared statement* std::string query = "SELECT \* FROM users WHERE username = ?"; sqlite3\_stmt \*stmt; sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, NULL); sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC); |

| Principle 1 - Validate Input Data:  This principle emphasizes the importance of validating data to prevent security vulnerabilities. SQL injection occurs when user input is improperly handled. By using parameterized queries or prepared statements, this standard ensures that input data is correctly validated and separated from SQL commands, preventing injection attacks and maintaining data integrity. |
| --- |
| Principle 7 - Sanitize Data Sent to Other Systems:  This principle involves cleaning and validating data before other systems process it. Preventing SQL injection aligns with this principle as parameterized queries or prepared statements ensure that data sent to the database is appropriately sanitized, protecting against malicious inputs and ensuring secure data handling. |
| Principle 8 - Practice Defense in Depth:  This principle involves implementing multiple layers of security controls. Preventing SQL injection through parameterized queries adds additional protection against unauthorized database access. By integrating such measures, developers ensure that even if one layer of defense fails, others are in place to protect the system. |
| Principle 10 - Adopt a Secure Coding Standard:  Following secure coding standards involves adhering to best practices to avoid common vulnerabilities. The standard of preventing SQL injection by using parameterized queries is a best practice in secure coding that helps protect against SQL injection attacks and ensures the security and integrity of the application. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **High** | **5** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [The Checker Framework](https://wiki.sei.cmu.edu/confluence/display/java/The+Checker+Framework) | 2.1.3 | **Tainting Checker** | The Tainting Checker in The Checker Framework helps detect potential SQL injection vulnerabilities by analyzing how tainted user inputs might affect SQL queries. It ensures that inputs are appropriately sanitized or parameterized before being used in SQL statements. |
| [Fortify](https://wiki.sei.cmu.edu/confluence/display/java/Fortify) | 1.0 | **HTTP\_Response\_Splitting** **SQL\_Injection\_\_Persistence** **SQL\_Injection** | Fortify's SQL\_Injection checkers detect potential SQL injection vulnerabilities by analyzing how data flows into SQL queries. It helps identify and remediate risks associated with improperly handled user inputs in SQL commands. |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/java/Klocwork) | 2024.2 | **SV.DATA.DB** **SV.SQL** **SV.SQL.DBSOURCE** | Klocwork's SV.DATA.DB, SV.SQL, and SV.SQL.DBSOURCE checkers analyze Code to detect SQL injection risks by identifying patterns where data from untrusted sources is used in SQL queries. They ensure that queries are parameterized and data is properly sanitized. |
| [SpotBugs](https://wiki.sei.cmu.edu/confluence/display/java/SpotBugs) | 4.6.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** **SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING** | SpotBugs' checkers SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE and SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING identify risks where SQL statements include non-constant strings, indicating potential SQL injection vulnerabilities. They help ensure that SQL queries are parameterized properly. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CLG | Do not access freed memory: Accessing memory after it has been freed can lead to undefined behavior, crashes, or data corruption. Ensuring that memory is not accessed after being freed prevents these issues and maintains program stability. |

| **Noncompliant Code** |
| --- |
| The Code attempts to access memory after it has been freed, which can lead to undefined behavior or program crashes. |
| int \*ptr = malloc(sizeof(int)); \*ptr = 10; free(ptr); \*ptr = 20; *// Accessing memory after it has been freed* |

| **Compliant Code** |
| --- |
| Set pointers to NULL after freeing memory to prevent access. |
| int \*ptr = malloc(sizeof(int)); \*ptr = 10; free(ptr); ptr = NULL; *// Prevents access to freed memory* |

| Principle 4 - Keep It Simple:  This principle advocates designing and writing straightforward Code to understand and reduce complexity. Ensuring that memory is not accessed after it has been freed helps maintain simplicity in memory management. The straightforward management of memory allocation and deallocation helps prevent errors and undefined behavior, enhances code stability, and reduces complexity. |
| --- |
| Principle 8 - Practice Defense in Depth:  Implementing safeguards against accessing freed memory adds additional protection to the system's stability. By ensuring that memory is not accessed after it has been freed, developers create multiple layers of defense against potential crashes or data corruption. This practice helps maintain system integrity even if other parts of the codebase have issues. |
| Principle 9 - Use Effective Quality Assurance Techniques:  Effective quality assurance techniques involve rigorous testing to identify and fix defects. Tools and practices that detect issues related to accessing freed memory are part of effective QA. By incorporating such tools, developers can catch and correct memory management issues before they lead to crashes or data corruption, ensuring a more robust and stable software application. |
| Principle 10 - Adopt a Secure Coding Standard:  Following secure coding standards involves following best practices for writing reliable and stable codes. The standard of not accessing freed memory aligns with secure coding practices that aim to prevent undefined behavior and ensure that the application remains stable and reliable. By adhering to this standard, developers avoid common pitfalls that can lead to crashes and data corruption. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **High** | **5** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM30** | Axivion's CertC-MEM30 checker identifies instances where memory may be accessed after being freed. It helps ensure that freed memory is not inadvertently used, preventing undefined behavior and potential crashes. |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2024.2 | **UFM.DEREF.MIGHT** **UFM.DEREF.MUST** **UFM.FFM.MIGHT** **UFM.FFM.MUST** **UFM.RETURN.MIGHT** **UFM.RETURN.MUST** **UFM.USE.MIGHT** **UFM.USE.MUST** | Klocwork's checkers for use-after-free (UFM.DEREF.\*, UFM.FFM.\*, UFM.RETURN.\*, UFM.USE.\*) detect potential dereferencing of freed memory and ensure that memory access patterns adhere to best practices, preventing access to deallocated memory. |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **449, 2434** | PC-lint Plus's checkers 449 and 2434 detect potential use-after-free issues by analyzing Code for access to deallocated memory. They help ensure that memory is handled correctly, preventing undefined behavior and crashes. |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **dangling\_pointer** | TrustInSoft Analyzer's dangling\_pointer checker identifies instances where memory might be accessed after being freed. It helps ensure that memory is not improperly used, preventing crashes and data corruption. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG | Never use assertions to validate method arguments: Assertions are a valuable tool for detecting programming errors and ensuring that assumptions and invariants hold true during development. By using assertions effectively, developers can catch errors early, maintain code correctness, and document expected conditions. However, assertions should not be used for handling runtime errors or user inputs, which should be managed through proper error checking and handling mechanisms. |

| **Noncompliant Code** |
| --- |
| The Code does not use assertions to validate that the pointer data is not NULL before dereferencing it, leading to potential runtime errors. |
| void process(int \*ptr) {  *// No assertion to check if ptr is not NULL* \*ptr = \*ptr + 1; }  int main() {  int \*data = NULL;  process(data); *// Potential dereference of NULL pointer* return 0; } |

| **Compliant Code** |
| --- |
| Use assertions to check for conditions that should be true in a correctly functioning program. |
| #include <assert.h>  void process(int \*ptr) {  assert(ptr != NULL); *// Assert that ptr is not NULL* \*ptr = \*ptr + 1; }  int main() {  int value = 10;  int \*data = &value;  process(data); *// Safe to process as ptr is not NULL* return 0; } |

| Principle 1 - Validate Input Data:  This principle emphasizes the importance of validating data before processing it to avoid vulnerabilities. Assertions should not be used for validating method arguments or user inputs, as they are typically only active in debug builds and may be disabled in production. Proper input validation should be implemented to ensure data integrity and prevent runtime errors or vulnerabilities from invalid data. |
| --- |
| Principle 3 - Architect and Design for Security Policies:  Architecting and designing with security policies in mind involves incorporating robust error-handling mechanisms. Relying on assertions for runtime error handling or input validation is insufficient. Instead, developers should design systems with comprehensive error-checking and handling mechanisms to ensure security and stability, as assertions alone do not provide adequate protection in all environments. |
| Principle 10 - Adopt a Secure Coding Standard:  Adopting secure coding standards involves following best practices for robust Code. This includes not using assertions for runtime error handling or validating method arguments, as secure coding practices require more reliable error-handling mechanisms. By adhering to secure coding standards, developers ensure that their Code can handle unexpected conditions and errors properly, maintaining stability and security. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | **Medium** | **3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S1006 | SonarQube's S1006 rule identifies the misuse of assertions for handling runtime errors or user inputs. It ensures that assertions are used correctly for internal consistency checks during development, while other error-handling mechanisms are used for user inputs and runtime errors. |
| Coverity | 2023.12 | CHECK\_ASSERT | Coverity's CHECK\_ASSERT checker helps identify incorrect use of assertions in Code. It ensures that assertions are used only for validating internal conditions during development and not for validating method arguments or handling runtime errors. |
| Clang Static Analyzer | 16.0 | assert-usage | Clang Static Analyzer's assert-usage checker detects misuse of assertions for method arguments and runtime errors. It ensures that assertions are used appropriately for internal validation, not user input validation or error handling. |
| PVS-Studio | 7.32 | V1017 | PVS-Studio's V1017 checker identifies incorrect use of assertions for method arguments or error handling. It helps ensure that assertions are used for development-time checks and not for managing runtime conditions or user inputs. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions: Proper handling of all exceptions ensures that unexpected errors are managed gracefully, preventing crashes and maintaining program stability. Ignoring exceptions can lead to unhandled errors and unreliable application behavior. |

| **Noncompliant Code** |
| --- |
| The Code throws an exception but does not provide a mechanism to handle it, leading to a potential program crash. |
| void foo() {  throw std::runtime\_error("An error occurred"); }  int main() {  foo(); *// Exception thrown but not handled* return 0; } |

| **Compliant Code** |
| --- |
| Implement exception handling to manage errors gracefully. |
| void foo() {  throw std::runtime\_error("An error occurred"); }  int main() {  try {  foo(); *// Handle the exception* } catch (const std::exception &e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  }  return 0; } |

| Principle 1 - Validate Input Data:  This principle underscores the importance of handling unexpected inputs securely. While it primarily focuses on validating data before processing, handling all exceptions also ensures that unexpected or invalid inputs are managed correctly, preventing crashes and maintaining system stability. Proper exception handling complements input validation by managing errors that arise from unforeseen issues during execution. |
| --- |
| Principle 3 - Architect and Design for Security Policies:  Handling all exceptions is critical to designing a system with robust security policies. By ensuring that all potential exceptions are caught and managed, developers can design systems that maintain stability and security, even in the face of unexpected errors. This principle integrates exception handling into the architecture to build more resilient and secure systems. |
| Principle 8 - Practice Defense in Depth:  Exception handling contributes to a defense-in-depth strategy by providing additional protection. Properly managing exceptions ensures that errors are contained and addressed without compromising the system's stability. This practice complements other security measures by adding robustness to the application's error management, reducing the likelihood of a complete system failure. |
| Principle 10 - Adopt a Secure Coding Standard:  Adopting a secure coding standard involves implementing best practices for error management, including handling all exceptions. This standard ensures that the Code is resilient to unexpected errors and maintains stability, aligning with secure coding practices that prevent unhandled exceptions from leading to crashes or unreliable behavior. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | **Low** | **2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.UCTCH** | CodeSonar's LANG.STRUCT.UCTCH checker analyzes Code to ensure all exceptions are caught and handled correctly. It detects areas where exceptions might not be handled, helping to prevent unhandled exceptions and ensuring program stability. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **C++4035, C++4036, C++4037** | Helix QAC's checkers C++4035, C++4036, and C++4037 enforce comprehensive exception handling in C++ code. They identify missing catch blocks and improper exception handling practices, helping to ensure that all exceptions are managed and the program remains reliable. |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.2 | **MISRA.CATCH.ALL** | Klocwork's MISRA.CATCH.ALL checker verifies that all exceptions are handled according to MISRA guidelines. It addresses every potential exception, improving code reliability by enforcing complete exception handling. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Polyspace Bug Finder's CERT C++: ERR51-CPP checker ensures that all exceptions are handled appropriately. It identifies missing catch blocks and promotes comprehensive exception handling to avoid crashes and maintain program stability. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object-Oriented Programming (OOP) | STD-008-CPP | Do not invoke virtual functions from constructors or destructors: Invoking virtual functions from constructors or destructors can lead to incomplete object states and undefined behavior, as the object's type may not be fully constructed or destructed. Ensuring that virtual functions are not called in these scenarios maintains object integrity. |

| **Noncompliant Code** |
| --- |
| In this example, the Base class constructor and destructor invoke the print virtual function. During the construction of a Derived object, the print function in the Base class constructor will be called before the Derived part is fully constructed, leading to unexpected behavior. Similarly, the Derived part will be destructed before the print function in the Base class destructor is called, which can also lead to undefined behavior. |
| #include <iostream>  class Base { public:  Base() {  *// Invoking virtual function in constructor* print();  }    virtual ~Base() {  *// Invoking virtual function in destructor* print();  }   virtual void print() const {  std::cout << "Base class print" << std::endl;  } };  class Derived : public Base { public:  Derived() : Base() {}    virtual void print() const override {  std::cout << "Derived class print" << std::endl;  } };  int main() {  Derived d;  return 0; } |

| **Compliant Code** |
| --- |
| Use non-virtual functions or ensure that virtual functions are not called during construction or destruction. |
| #include <iostream>  class Base { public:  Base() {  *// Safe operation: No virtual functions called* initialize();  }    virtual ~Base() {  *// Safe operation: No virtual functions called* cleanup();  }   void initialize() {  std::cout << "Base class initialization" << std::endl;  }   void cleanup() {  std::cout << "Base class cleanup" << std::endl;  }   virtual void print() const {  std::cout << "Base class print" << std::endl;  } };  class Derived : public Base { public:  Derived() : Base() {}   virtual void print() const override {  std::cout << "Derived class print" << std::endl;  } };  int main() {  Derived d;  d.print(); *// Safe to call virtual function now* return 0; } |

| Principle 4 - Keep It Simple:  This principle advocates straightforwardly designing systems to reduce complexity and avoid potential issues. By avoiding invading virtual functions from constructors or destructors, developers simplify the object lifecycle and ensure that virtual calls do not lead to undefined behavior. This practice supports simplicity by preventing complex interactions arising from incomplete object states, making the Code more predictable and easier to understand. |
| --- |
| Principle 8 - Practice Defense in Depth:  This principle involves implementing multiple layers of security to protect systems. By ensuring that virtual functions are not called during object construction or destruction, developers add a layer of defense against undefined behavior and potential security vulnerabilities. This practice supports overall system robustness by preventing subtle bugs and ensuring that object states are fully defined before invoking virtual methods. |
| Principle 10 - Adopt a Secure Coding Standard:  Secure coding standards include following best practices, such as not invoking virtual functions from constructors or destructors. This standard helps avoid undefined behavior and maintains object integrity, aligning with secure coding practices that prevent vulnerabilities and ensure predictable and stable Code. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **Low** | **1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Klocwork) | 2024.2 | **CERT.OOP.CTOR.VIRTUAL\_FUNC** | Klocwork's CERT.OOP.CTOR.VIRTUAL\_FUNC checker focuses on preventing the invocation of virtual functions from constructors. This helps to ensure that the object is fully constructed before any virtual functions are called, avoiding potential issues related to incomplete object states. |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **467 S, 92 D** | LDRA's checkers 467 S and 92 D identify improper calls to virtual functions during object construction and destruction. They ensure that such calls are detected and reported, helping to maintain object integrity and prevent undefined behavior. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **virtual-call-in-constructor** | RuleChecker's virtual-call-in-constructor checker identifies instances where virtual functions are called from constructors. It helps enforce best practices by ensuring that virtual functions are not invoked before the object is fully constructed, avoiding issues related to incomplete object states and undefined behavior. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [S1699](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-1699) | SonarQube's S1699 checker detects virtual function calls within constructors or destructors. It helps ensure that virtual functions are only invoked when the object is entirely stable. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | STD-009-CLG | Clean up thread-specific storage: Failing to clean up thread-specific storage can lead to resource leaks and unpredictable behavior. Ensuring that thread-specific storage is properly cleaned up prevents these issues and maintains system stability. |

| **Noncompliant Code** |
| --- |
| In this example, the thread-specific storage is created and used, but it is not properly cleaned up. The destructor function for pthread\_key\_delete is not set, so the allocated memory for thread-specific storage is not freed, leading to potential memory leaks. |
| #include <pthread.h> #include <stdio.h>  pthread\_key\_t key;  void\* thread\_func(void\* arg) {  int\* thread\_data = (int\*)pthread\_getspecific(key);  if (thread\_data == NULL) {  thread\_data = (int\*)malloc(sizeof(int));  \*thread\_data = 42;  pthread\_setspecific(key, thread\_data);  }  printf("Thread-specific data: %d**\n**", \*thread\_data);  return NULL; }  int main() {  pthread\_t threads[2];    *// Initialize thread-specific key* pthread\_key\_create(&key, NULL);   *// Create threads* for (int i = 0; i < 2; ++i) {  pthread\_create(&threads[i], NULL, thread\_func, NULL);  }   *// Join threads* for (int i = 0; i < 2; ++i) {  pthread\_join(threads[i], NULL);  }   *// Thread-specific storage not cleaned up* pthread\_key\_delete(key);  return 0; } |

| **Compliant Code** |
| --- |
| Ensure proper cleanup of thread-specific storage by providing a destructor function to pthread\_key\_create that will be called to release resources when a thread-specific key is deleted. |
| #include <pthread.h> #include <stdio.h> #include <stdlib.h>  pthread\_key\_t key;  void cleanup(void\* data) {  free(data); }  void\* thread\_func(void\* arg) {  int\* thread\_data = (int\*)pthread\_getspecific(key);  if (thread\_data == NULL) {  thread\_data = (int\*)malloc(sizeof(int));  \*thread\_data = 42;  pthread\_setspecific(key, thread\_data);  }  printf("Thread-specific data: %d**\n**", \*thread\_data);  return NULL; }  int main() {  pthread\_t threads[2];    *// Initialize thread-specific key with cleanup function* pthread\_key\_create(&key, cleanup);   *// Create threads* for (int i = 0; i < 2; ++i) {  pthread\_create(&threads[i], NULL, thread\_func, NULL);  }   *// Join threads* for (int i = 0; i < 2; ++i) {  pthread\_join(threads[i], NULL);  }   *//cleanup thread-specific key* pthread\_key\_delete(key);  return 0; } |

| Principle 4 - Keep It Simple:  Properly cleaning up thread-specific storage simplifies resource management by properly releasing all allocated resources. This reduces complexity and prevents potential issues related to resource leaks, which can complicate debugging and maintenance. By adhering to this principle, developers keep the system's resource management straightforward and reliable. |
| --- |
| Principle 8 - Practice Defense in Depth:  Cleaning up thread-specific storage contributes to a defense-in-depth strategy by ensuring that resource management issues are minimized even if other layers of security fail. Proper cleanup prevents resource leaks and maintains system stability, adding a layer of robustness to the overall security posture of the application. |
| Principle 9 - Use Effective Quality Assurance Techniques:  Effective quality assurance includes rigorous testing of resource management practices, including the cleanup of thread-specific storage. By thoroughly testing and validating that thread-specific storage is appropriately managed, developers ensure that the software behaves as expected and does not suffer from resource leaks or unpredictable behavior, enhancing overall software quality and security. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | **Low** | **2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **ALLOC.LEAK** | CodeSonar's ALLOC.LEAK checker identifies potential memory leaks, including those that may arise from thread-specific storage. This checker helps ensure that allocated resources are properly freed, preventing resource leaks and ensuring stability. |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **ALLOC\_FREE\_MISMATCH** | Coverity's ALLOC\_FREE\_MISMATCH checker detects mismatches between allocation and deallocation calls, which includes ensuring that thread-specific storage is properly cleaned up. This helps prevent resource leaks by verifying that all allocated memory is appropriately freed. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.2 | **C1780, C1781, C1782, C1783, C1784** | Helix QAC's set of checkers for thread-specific storage focuses on ensuring that all resources, including those specific to threads, are properly managed and cleaned up. These checkers help prevent resource leaks and maintain system stability. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-CON30-a** | Parasoft C/C++test's CERT\_C-CON30-a checker identifies issues related to resource management, including the cleanup of thread-specific storage. It helps ensure that all allocated resources are properly released. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-010-CPP | Guarantee exception safety: Guaranteeing exception safety ensures that resources are properly managed and the program remains in a consistent state even when exceptions occur. This prevents resource leaks and maintains program stability. |

| **Noncompliant Code** |
| --- |
| This Code does not guarantee exception safety because it manually manages the resource cleanup and can lead to resource leaks if an exception occurs before the cleanup code is executed. |
| #include <iostream> #include <memory> #include <stdexcept>  class DatabaseConnection { public:  DatabaseConnection() {  *// Code to open the database connection* std::cout << "Database connection opened.**\n**";  }   ~DatabaseConnection() {  *// Code to close the database connection* std::cout << "Database connection closed.**\n**";  }   void executeQuery() {  *// Simulate a query that might throw an exception* throw std::runtime\_error("Query failed");  } };  void performDatabaseOperation() {  DatabaseConnection\* dbConn = new DatabaseConnection();  try {  dbConn->executeQuery();  *// Additional operations* } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << '**\n**';  *// Resource cleanup is manual and might be skipped if an exception occurs* delete dbConn;  throw; *// Re-throw the exception* }  delete dbConn; *// Resource cleanup* }  int main() {  try {  performDatabaseOperation();  } catch (const std::exception& e) {  std::cerr << "Unhandled Error: " << e.what() << '**\n**';  }  return 0; } |

| **Compliant Code** |
| --- |
| This Code uses Resource Acquisition Is Initialization (RAII) to manage the DatabaseConnection resource, ensuring that it is properly cleaned up regardless of whether an exception occurs. |
| #include <iostream> #include <memory> #include <stdexcept>  class DatabaseConnection { public:  DatabaseConnection() {  *// Code to open the database connection* std::cout << "Database connection opened.**\n**";  }   ~DatabaseConnection() {  *// Code to close the database connection* std::cout << "Database connection closed.**\n**";  }   void executeQuery() {  *// Simulate a query that might throw an exception* throw std::runtime\_error("Query failed");  } };  void performDatabaseOperation() {  *// Using smart pointer for RAII* std::unique\_ptr<DatabaseConnection> dbConn = std::make\_unique<DatabaseConnection>();  try {  dbConn->executeQuery();  *// Additional operations* } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << '**\n**';  *// No need for manual cleanup, dbConn will be automatically cleaned up* throw; *// Re-throw the exception* }  *// No need to explicitly delete dbConn; it will be automatically cleaned up* }  int main() {  try {  performDatabaseOperation();  } catch (const std::exception& e) {  std::cerr << "Unhandled Error: " << e.what() << '**\n**';  }  return 0; } |

| Principle 3 - Architect and Design for Security Policies:  Guaranteeing exception safety is integral to designing a system with robust security policies. By architecting the system to handle exceptions gracefully and ensuring that resources are appropriately managed even when exceptions occur, developers create a resilient design to unexpected errors. This principle emphasizes incorporating error handling as a fundamental aspect of system architecture. |
| --- |
| Principle 4 - Keep It Simple:  Ensuring exception safety simplifies error management by maintaining resource handling and program state consistency. A straightforward approach to managing exceptions and resources reduces complexity and minimizes the potential for bugs related to exception handling, thereby making the system easier to understand and maintain. |
| Principle 8 - Practice Defense in Depth:  Guaranteeing exception safety is a defensive measure that adds an extra layer of protection against unexpected failures. Ensuring that resources are correctly managed, even during exceptions, aligns with the defense-in-depth strategy of having multiple layers of security to maintain system stability and prevent resource leaks. |
| Principle 9 - Use Effective Quality Assurance Techniques:  Effective QA techniques involve rigorous testing to ensure that the system maintains exception safety. By validating that resources are appropriately managed and the program remains consistent despite exceptions, developers use QA processes to confirm that the system adheres to exception safety standards, enhancing reliability and security. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | **Medium** | **4** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.LEAK** | CodeSonar's ALLOC.LEAK checker identifies potential memory leaks, which can be crucial for exception safety. It helps ensure that resources are properly released even when exceptions occur, preventing leaks and maintaining a stable program state. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **C++4075, C++4076** | Helix QAC's checkers C++4075 and C++4076 focus on exception handling and resource management. They help ensure that exceptions are properly handled and resources are consistently managed, thereby supporting exception safety in your Code. |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **527 S, 56 D, 71 D** | LDRA's checkers for exception handling include 527 S, 56 D, and 71 D, which help ensure that resources are correctly managed and released in the event of exceptions. These checkers support maintaining a stable and consistent program state. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.32 | [V565](https://pvs-studio.com/en/docs/warnings/v565/)**,**[V1023](https://pvs-studio.com/en/docs/warnings/v1023/)**,**[V5002](https://pvs-studio.com/en/docs/warnings/v5002/) | PVS-Studio's checkers V565, V1023, and V5002 focus on identifying potential issues related to exception safety and resource management. They help ensure that resources are not leaked and that the program state remains stable when exceptions are thrown. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



### Automation



Automation should be integrated throughout Green Pace's DevSecOps pipeline to ensure compliance with the security standards defined in this policy. In the pre-production phase, automation tools are essential in the "Assess and Plan" stage to monitor the threat landscape and regulatory changes continuously. Tools like static analysis and dependency checkers should be utilized to detect vulnerabilities early in the process. During the "Design" phase, automated tools can enforce security best practices, such as those outlined by OWASP, ensuring that security considerations are built into the design. The "Build" stage should leverage automation to secure the build process by validating components through trusted repositories and digitally signed artifacts. In the "Verify and Test" phase, automated security testing, including vulnerability scanning and compliance checks, should be mandatory to detect and address issues before moving code to production.

In the production phase, automation continues to play a crucial role. During the "Transition and Health Check" stage, automated tools should configure and deploy security settings while performing penetration testing to ensure the environment is secure before going live. In "Monitor and Detect," automation is critical for continuous security monitoring, using systems like SIEM for real-time threat detection. Automated responses in the "Respond" phase can mitigate risks by blocking attacks and reverting to stable states if necessary. Finally, in the "Maintain and Stabilize" stage, automated tools should continuously assess the system against security baselines, ensuring that any deviations are corrected promptly to maintain a secure and stable environment. By embedding automation throughout these stages, Green Pace can effectively enforce security policies, maintain compliance, and strengthen its overall security posture.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-002-CPP | Medium | Probable | Medium | Medium | 3 |
| STD-003-CLG | Low | Likely | Low | Medium | 3 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CLG | High | Likely | Medium | High | 5 |
| STD-006-CLG | Medium | Probable | Medium | Medium | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 2 |
| STD-008-CPP | Low | Unlikely | Medium | Low | 1 |
| STD-009-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-010-CPP | High | Likely | High | Medium | 4 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects data stored on physical media (like hard drives or cloud storage) from unauthorized access. This encryption ensures that the data remains unreadable and secure even if unauthorized parties access storage media. The policy for encryption at rest should apply to all sensitive data, including personal information, financial data, and intellectual property. The policy mandates that all stored data be encrypted using robust algorithms such as AES-256 to prevent data breaches and unauthorized access, comply with regulatory requirements, and protect the organization's assets. |
| Encryption in flight | Encryption in flight (also known as data in transit) involves securing data as it moves across networks, whether between devices, servers, or clients. This type of encryption ensures that any intercepted data during transmission cannot be read or altered by attackers. The policy for encryption in flight applies to all data transmitted over public or internal networks, particularly sensitive data shared between users, systems, or external partners. The policy should enforce secure communication protocols like TLS/SSL to safeguard against eavesdropping, man-in-the-middle attacks, and data tampering during transmission. |
| Encryption in use | Encryption in use refers to protecting data while it is being processed, such as in memory or CPU registers. This is crucial for safeguarding data from exposure to unauthorized processes or users, especially in environments where data is accessed in real time, such as in financial transactions or sensitive computations. The policy for encryption in use should apply to all critical applications and data processing environments. It mandates using technologies like homomorphic encryption or secure enclaves to ensure that sensitive data remains protected even during active processing, reducing the risk of memory scraping or other in-use data vulnerabilities. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is verifying a user's or system's identity before granting access to resources. This ensures that only authorized individuals or systems can access sensitive data or perform critical operations. The policy for authentication should include the requirement for secure user logins, employing multi-factor authentication to enhance security. This policy should apply to all systems and applications requiring identity verification, preventing unauthorized access and mitigating potential security risks. The organization reduces the risk of impersonation and unauthorized access by ensuring robust authentication measures. |
| Authorization | Authorization determines an authenticated user or system's actions based on their roles and responsibilities. The policy for authorization should specify permissions for user level of access, including changes to the database and the addition of new users. It should clearly define what each user or role can access and modify, including the files users access. This policy helps to prevent unauthorized actions, ensuring that users only have access to the resources necessary for their roles, and provides a clear framework for managing permissions and access controls. |
| Accounting | Accounting involves tracking and recording user activities to ensure compliance with security policies and to detect potential security incidents. This includes comprehensive logging of user logins, changes to the database, addition of new users, and files accessed by users. The policy for accounting should mandate detailed logging and periodic review of these logs to identify and address any anomalies or security concerns. This helps to provide an audit trail, support forensic investigations, and ensure adherence to security policies, enhancing overall accountability and transparency. |

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum | Matthew Rearick |
| 2.0 | 08/11/2024 | Added 10 Security Principals  Added 10 Coding Standards   * Type * Label * Description * Noncompliant & Compliant Examples * Relevant Principals * Threat Assessment * Automation Tools   Added DevSecOps Automation Integration Guidelines  Added Risk Assessment Summary  Added Encryption and Triple-A Policies  Removed Template Guidelines for Previously Incomplete Sections | Matthew Rearick |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |

# References

*Acronis. (2024). 2024 Cybersecurity Trends: Key steps, strategies and guidance. Retrieved from https://www.acronis.com/en-us/blog/posts/cyber-security-trends/*

*Harris, S. (2024). Cybersecurity Gap Analysis: Brief Overview and Insights. Journal of Cybersecurity, 14(2), 45-60. Retrieved from https://www.threatintelligence.com/blog/cybersecurity-gap-analysis*

*Smith, J. (2024). The 10 Biggest Cyber Security Trends In 2024 Everyone Must Be Ready For Now. Cybersecurity Today, 21(3), 77-85 Retrieved from https://www.forbes.com/sites/bernardmarr/2023/10/11/the-10-biggest-cyber-security-trends-in-2024-everyone-must-be-ready-for-now/*