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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## 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 | All input data must be validated when coming from untrusted sources before processing. These efforts are to prevent malicious data from causing intended behavior, including, but not limited to injection attacks & buffer overflows. This ensures data & application security & integrity. |
| 1. Heed Compiler Warnings | The compiler warnings indicate potential issues in the code that can lead to security vulnerabilities. The Developer should look at the warnings as errors & quickly work to resolve them to prevent undefined behaviors from reaching published code. |
| 1. Architect and Design for Security Policies | Address security during the architecture and design phases, not after. By planning for it from the start, you can build stronger systems that are less vulnerable to attacks from the very beginning. |
| 1. Keep It Simple | Favor simple code and system designs over complex ones. They are less prone to errors, easier to audit for security gaps, and more robust against exploitation. |
| 1. Default Deny | Implement the principle of least privilege by default. Deny all access initially and grant permissions explicitly based on strict need-to-know criteria. This approach minimizes exposure and reduces the potential impact of a security breach. |
| 1. Adhere to the Principle of Least Privilege | Enforce the principle of least privilege. By granting only the minimum access necessary for a specific task, you minimize the potential damage from both accidental errors and malicious attacks, thereby strengthening security. |
| 1. Sanitize Data Sent to Other Systems | [Implement rigorous output encoding and data sanitization for all inter-system communications. This practice ensures the data is inert and safe to process, neutralizing injection-based threats before they can reach downstream systems. |
| 1. Practice Defense in Depth | Adopt a layered security architecture. By employing redundant and diverse defensive measures at each system level, you create a series of barriers that contain breaches and provide fail-safe protection against single points of failure. |
| 1. Use Effective Quality Assurance Techniques | Proactively hunt for vulnerabilities by mandating code reviews, implementing automated static and dynamic testing tools, and conducting thorough QA tests. Catching defects in development prevents them from becoming security incidents in production. |
| 1. Adopt a Secure Coding Standard | Follow established & pre-set coding standards to ensure consistency & apply best practices. This will cutdown the chances of introducing new security flaws due to coding errors, mistypes, & block misunderstandings. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

**Coding Standard 1**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Avoid implicit type conversions.  - Implicit type conversions can lead to unexpected behavior, data loss, or security vulnerabilities. By avoiding implicit conversions, we ensure data integrity and prevent subtle bugs. |

| **Noncompliant Code** |
| --- |
| **Implicit conversion from a larger type to a smaller type can cause data loss.** |
| unsigned long largeValue = 5000000000;  unsigned int smallValue = largeValue; // Implicit conversion may truncate data |

| **Compliant Code** |
| --- |
| **Explicitly handle type conversion to ensure data integrity.** |
| unsigned long largeValue = 5000000000;  if (largeValue <= UINT\_MAX) {  unsigned int smallValue = static\_cast<unsigned int>(largeValue);  // Proceed with smallValue  } else {  // Handle the error condition  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Input data should be rigorously validated to safeguard its integrity and prevent errors stemming from incorrect data type conversions. 2. Developers must treat compiler warnings as errors, paying special attention to those highlighting type mismatches to ensure problems are fixed early. 3. A comprehensive quality assurance process, including unit tests and peer reviews, is essential for uncovering unintended type conversions that may not be immediately obvious. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S1871 | Detects implicit type conversions in C++ code to ensure strong type safety. |
| Clang-Tidy | 15.0 | readability-implicit-cast | Flags instances of implicit type conversions, ensuring code clarity and safety. |
| Coverity | 2023.12 | Type Mismatch Checker | Identifies potential mismatches and implicit conversions that could lead to errors |
| Cppcheck | 2.10 | typeCastCheck | Highlights implicit or unsafe typecasting in C++ projects. |

**Coding Standard 2**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Check the return values of functions.  - Ignoring return values of functions, especially those that report errors, can lead to unexpected behavior and security vulnerabilities. Always check return values to handle errors appropriately. |

| **Noncompliant Code** |
| --- |
| **Fails to check the return value of fgets, which may result in using uninitialized data.** |
| char buffer[256];  fgets(buffer, sizeof(buffer), stdin);  // Assumes buffer contains valid data  processInput(buffer); |

| **Compliant Code** |
| --- |
| **Checks the return value of fgets before using the data.** |
| char buffer[256];  if (fgets(buffer, sizeof(buffer), stdin) != NULL) {  processInput(buffer);  } else {  // Handle error condition  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Ensure functions succeed by checking their return values, preventing unexpected behavior from unhandled failures. 2. Use the compiler's diagnostics to catch instances where return values are ignored. 3. Incorporate static and dynamic analysis tools into the testing process to automatically detect unhandled return values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S3626 | Detects unhandled return values in function calls. |
| Clang-Tidy | 15.0 | misc-ignored-return-value | Flags calls to functions whose return values are ignored. |
| Coverity | 2023.12 | Return Value Check | Identifies unhandled function return values that may lead to security issues. |
| Cppcheck | 2.10 | checkReturn | Highlights ignored return values for critical functions in C/C++ codebases. |

**Coding Standard 3**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP | Properly null-terminate strings.  - Failure to null-terminate strings can result in buffer overflows and security vulnerabilities. Ensure all strings are properly null-terminated to prevent reading or writing beyond the intended memory. |

| **Noncompliant Code** |
| --- |
| **Uses strncpy without ensuring null termination, which may result in non-null-terminated strings.** |
| char dest[10];  strncpy(dest, src, sizeof(dest));  // dest may not be null-terminated |

| **Compliant Code** |
| --- |
| **Ensures the destination string is null-terminated.** |
| char dest[10];  strncpy(dest, src, sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Sanitize and validate all string inputs to enforce correct formatting and termination, preventing buffer overflows. 2. Always null-terminate strings to simplify memory management and avoid common errors. 3. Layer multiple security defenses to contain vulnerabilities arising from improper memory access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S5537 | Identifies cases where strings may not be null-terminated. |
| Clang-Tidy | 15.0 | bugprone-string-constructor | Flags improper string handling and termination issues. |
| Coverity | 2023.12 | Buffer Overrun Checker | Detects improper memory handling, including string null-termination issues. |
| Cppcheck | 2.10 | nullTermination | Identifies strings that may not be null-terminated, preventing memory errors. |

**Coding Standard 4**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Use prepared statements for database queries.  - Using prepared statements with parameterized queries helps prevent SQL injection attacks by separating SQL code from data. |

| **Noncompliant Code** |
| --- |
| **Concatenates user input directly into SQL query, making it vulnerable to SQL injection.** |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  executeQuery(query); |

| **Compliant Code** |
| --- |
| **Uses a prepared statement to safely include user input.** |
| std::string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement\* stmt = conn->prepareStatement(query);  stmt->setString(1, username);  ResultSet\* rs = stmt->executeQuery(); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Sanitize all inputs using prepared statements to separate data from commands. 2. Whitelist permitted SQL operations to block unauthorized database actions. 3. Layer security measures to ensure protection even if one control fails. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S3649 | Detects SQL queries that are vulnerable to injection due to string concatenation. |
| Clang-Tidy | 15.0 | bugprone-sql-injection | Flags SQL queries constructed using unsafe string concatenation. |
| Coverity | 2023.12 | SQL Injection Checker | Identifies potential SQL injection vulnerabilities in code. |
| OWASP ZAP | 2.13.0 | SQL Injection Scanner | Scans web applications for SQL injection vulnerabilities. |

**Coding Standard 5**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Free up available dynamic memory.  - Failing to release dynamically allocated memory leads to memory leaks, which can exhaust system resources and lead to application failure or denial of service. |

| **Noncompliant Code** |
| --- |
| **Allocates memory but never releases it.** |
| int\* data = new int[100];  // ... use data  // Memory is not freed |

| **Compliant Code** |
| --- |
| **Releases allocated memory when it is no longer needed.** |
| int\* data = new int[100];  // ... use data  delete[] data;  data = nullptr; |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Layer memory protections to mitigate leaks, corruption, and exhaustion. 2. Use automated analysis tools to detect memory leaks during development and testing. 3. Enforce secure memory management rules to ensure consistent and safe code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | Memcheck | Detects memory leaks and improper memory use in C++ programs. |
| SonarQube | 9.6 | cpp:S1751 | Identifies cases of unreleased dynamically allocated memory. |
| Clang-Tidy | 15.0 | cert-mem54-cpp | Checks for memory leaks and proper cleanup of dynamically allocated resources. |
| Coverity | 2023.12 | Resource Leak Checker | Flags dynamically allocated memory that is not released. |

**Coding Standard 6**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Do not use assertions for the handling of errors.  - Assertions are meant for detecting programming errors during development, not for handling runtime errors in production code. Using assertions for error handling may cause unexpected termination. |

| **Noncompliant Code** |
| --- |
| **Uses assert to check for runtime errors, which can be disabled and may not handle errors appropriately.** |
| int result = doSomething();  assert(result != -1);  // Proceed assuming success |

| **Compliant Code** |
| --- |
| **Checks for errors and handles them appropriately.** |
| int result = doSomething();  if (result == -1) {  // Handle error  } else {  // Proceed with result  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Resolve all compiler warnings to eliminate potential runtime errors that assertions won't catch. 2. Write simple, predictable error-handling code to improve maintainability and reliability. 3. Test error conditions comprehensively using QA tools, ensuring checks are active in production, unlike assertions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-err34-cpp | Detects misuse of assertions for error handling in C++ code. |
| SonarQube | 9.6 | cpp:S3603 | Identifies assertion misuse and recommends proper error handling methods. |
| Cppcheck | 2.10 | assertCheck | Flags inappropriate use of assertions for runtime error handling. |
| Coverity | 2023.12 | Assertion Checker | Highlights improper use of assertions that may fail in production environments. |

**Coding Standard 7**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Label catch exceptions by reference.  - Catching exceptions by value can lead to slicing and inefficient copies. Catching by reference ensures that the complete exception object is caught and handled. |

| **Noncompliant Code** |
| --- |
| **Catches exceptions by value, which may result in object slicing.** |
| try {  // Code that may throw  } catch (MyException e) {  // Handle exception  } |

| **Compliant Code** |
| --- |
| **Catches exceptions by const reference to avoid slicing.** |
| try {  // Code that may throw  } catch (const MyException& e) {  // Handle exception  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Catch exceptions by const reference to avoid object slicing and preserve type information. 2. Test all exception paths to identify and fix incomplete or inefficient error handling. 3. Enforce a consistent exception handling policy across the codebase for reliability and clarity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S1181 | Flags exception handling by value and recommends catching by reference. |
| Clang-Tidy | 15.0 | modernize-avoid-catch-by-value | Detects instances of catching exceptions by value and suggests using references. |
| Cppcheck | 2.10 | exceptionCheck | Highlights inefficient or unsafe exception handling practices. |
| Coverity | 2023.12 | Exception Handling Checker | Identifies improper exception handling, such as catching by value. |

**Coding Standard 8**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Resource Management** | STD-008-CPP | Release resources in all available paths.  - Failing to release resources like file handles or network connections can lead to resource exhaustion. Ensure that resources are released in all execution paths, including exceptions. |

| **Noncompliant Code** |
| --- |
| Does not release file handle if an exception is thrown. |
| std::ifstream file("data.txt");  // Perform operations that may throw exceptions  processFile(file);  // File is not closed if an exception occurs |

| **Compliant Code** |
| --- |
| **Uses RAII (Resource Acquisition Is Initialization) to ensure the file is closed even if an exception is thrown.** |
| void processFile(const std::string& filename) {  std::ifstream file(filename);  if (!file) {  // Handle error  }  // File will be closed automatically when going out of scope  process(file);  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. Layer resource protections to mitigate the risk of exhaustion-based attacks. 2. Use RAII for all resources to automate cleanup and prevent leaks. 3. Enforce a resource management standard that mandates RAII for consistency and reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | MemCheck | Detects resource leaks, such as unclosed file handles and memory leaks. |
| SonarQube | 9.6 | cpp:S2095 | Flags unclosed resources like file handles or sockets. |
| Clang-Tidy | 15.0 | cert-err52-cpp | Identifies missing cleanup for dynamically or manually managed resources. |
| Coverity | 2023.12 | Resource Leak Checker | Identifies potential resource leaks in all execution paths, including exceptions. |

**Coding Standard 9**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Error Handling** | STD-009-CPP | Do not ignore any exceptions.  - Ignoring exceptions can lead to undefined behavior and security vulnerabilities. All exceptions should be appropriately handled to maintain application stability. |

| **Noncompliant Code** |
| --- |
| **Catches exceptions but does nothing, hiding errors.** |
| try {  performCriticalOperation();  } catch (const std::exception&) {  // Silently ignore the exception  } |

| **Compliant Code** |
| --- |
| **Handles the exception or rethrows it after logging.** |
| try {  performCriticalOperation();  } catch (const std::exception& e) {  // Handle exception appropriately  logError(e.what());  throw; // Rethrow exception if it cannot be handled here  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. **Log every exception for analysis and troubleshooting.** 2. **Avoid silent failures with explicit and visible error handling.** 3. **Use exceptions as a stability layer to contain unexpected errors.** |
| --- |

**Threat Level**

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S2737 | Detects empty exception handlers that fail to process or log errors. |
| Clang-Tidy | 15.0 | bugprone-empty-catch | Flags catch blocks that do not properly handle or log exceptions. |
| Cppcheck | 2.10 | exceptionHandling | Identifies exception handlers that silently ignore exceptions without action. |
| Coverity | 2023.12 | Exception Handling Checker | Highlights cases where exceptions are caught but not processed effectively. |

**Coding Standard 10**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Concurrency** | STD-010-CPP | Avoid racing data in multi-threaded codes.  - Data races occur when multiple threads access shared data without proper synchronization, leading to undefined behavior. Use synchronization mechanisms to prevent data races. |

| **Noncompliant Code** |
| --- |
| Accesses shared data without synchronization. |
| int sharedCounter = 0;  void incrementCounter() {  for (int i = 0; i < 1000; ++i) {  ++sharedCounter; // Data race  }  } |

| **Compliant Code** |
| --- |
| Uses mutexes to synchronize access to shared data. |
| #include <mutex>  int sharedCounter = 0;  std::mutex counterMutex;  void incrementCounter() {  for (int i = 0; i < 1000; ++i) {  std::lock\_guard<std::mutex> lock(counterMutex);  ++sharedCounter; // Safe access  }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1. **Layer synchronization mechanisms** to safeguard shared data. 2. **Use stress testing and analysis tools** to uncover concurrency bugs. 3. **Enforce a thread-safe coding standard** to prevent data races and ensure consistency. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | Integrated into GCC/Clang | Data Race Detector | Dynamically detects data races in multithreaded applications during runtime. |
| Valgrind | 3.21.0 | Helgrind | Identifies data races and synchronization issues in C++ programs. |
| Clang-Tidy | 15.0 | cert-msc01-cpp | Flags shared variables accessed without proper synchronization mechanisms. |
| Coverity | 2023.12 | Concurrency Checker | Identifies data races and improper thread synchronization. |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Insert your written explanations here.]

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

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

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| **Encryption at rest** | * **Definition: Protects stored data by encrypting it using secure algorithms.** * **Application: Applied to databases, backups, and file systems storing sensitive data.** * **Purpose: Prevent unauthorized access to data in case of physical theft or system compromise.** |
| **Encryption in flight** | * **Definition: Secures data transmitted across networks.** * **Application: Used for securing HTTP (HTTPS), emails (TLS), and API communications.** * **Purpose: Prevents interception of sensitive data during transmission (e.g., man-in-the-middle attacks).** |
| **Encryption in use** | * **Definition: Encrypts data while being processed in memory.** * **Application: Applied to applications handling sensitive data like cryptographic keys or personal data.** * **Purpose: Prevents data leakage through memory dumps or unauthorized system access.** |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| **Authentication** | * **Definition: Verifies the identity of users accessing the system.** * **Application: Implement strong multi-factor authentication for user logins and privileged operations.** * **Purpose: Prevent unauthorized access to critical resources.** |
| **Authorization** | * **Definition: Grants or denies user actions based on roles or policies.** * **Application: Use role-based access control (RBAC) to restrict user privileges to the minimum necessary.** * **Purpose: Ensures that users only access resources necessary for their role.** |
| **Accounting** | * **Definition: Tracks and logs user activities for auditing and compliance.** * **Application: Log all login attempts, data changes, and resource accesses.** * **Purpose: Enables traceability and accountability for security audits and incident investigations.** |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

|  |  |  |
| --- | --- | --- |
| **Standard ID** | **Principles** | **Justification** |
| **STD-001-CPP** | **Validate Input Data Heed Compiler Warnings** | **Prevents type safety violations by sanitizing input and resolving type-related warnings.** |
| **STD-002-CPP** | **Validate Input Data Use Effective QA** | **Ensures proper handling of function return values to prevent undefined behavior, verified through testing.** |
| **STD-003-CPP** | **Validate Input Data Practice Defense in Depth** | **Secures string handling by enforcing proper formatting and termination to prevent buffer overflow vulnerabilities.** |
| **STD-004-CPP** | **Validate Input Data, Default Deny** | **Uses parameterized queries to prevent SQL injection by separating data from commands.** |
| **STD-005-CPP** | **Practice Defense in Depth Use Effective QA** | **Prevents resource exhaustion and memory leaks through layered safeguards and verification.** |
| **STD-006-CPP** | **Heed Compiler Warnings Keep It Simple** | **Replaces unreliable assertions with explicit error handling for production-ready stability.** |
| **STD-007-CPP** | **Use Effective QA**  **Keep It Simple** | **Prevents exception object slicing (by catching by reference) for correct error propagation and analysis.** |
| **STD-008-CPP** | **Practice Defense in Depth, Keep It Simple** | **Guarantees resource cleanup and prevents leaks using RAII for automatic, exception-safe management.** |
| **STD-009-CPP** | **Use Effective QA**  **Practice Defense in Depth** | **Logs and analyzes exceptions to diagnose failures and maintain application stability under error conditions.** |
| **STD-010-CPP** | **Practice Defense in Depth**  **Use Effective QA** | **Prevents data races and ensures thread safety through synchronization, verified by rigorous analysis.** |

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## 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 |  |
| 1.1 | 09/21/2025 | Milestone One. | Jamar Sampson |  |
| 1.2 | 10/11/2025 | Project One. | Jamar Sampson |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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