**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 | Input validation ensures that all data entered into a system is validated against expected formats and values. By validating inputs, developers can mitigate risks such as SQL injection and buffer overflows by preventing malicious or unexpected data from being processed. |
| 1. Heed Compiler Warnings | Pay attention to and resolve compiler warnings, as they often highlight potential vulnerabilities or bad practices in your code. By addressing these warnings, you can identify and fix issues such as type mismatches, memory leaks, and uninitialized variables before they lead to security problems. |
| 1. Architect and Design for Security Policies | Incorporate security considerations into the system’s architecture and design from the outset. Establishing clear security policies ensures that applications are built to resist threats, meet compliance requirements, and maintain a consistent level of protection as they scale or evolve. |
| 1. Keep It Simple | Simpler systems are easier to understand, audit, and secure. Avoid unnecessary complexity in your code and designs, as complex systems are more prone to errors and vulnerabilities. It’s much easier to defend a smaller fort than a larger one. |
| 1. Default Deny | Deny access to resources and actions by default unless explicitly allowed. This principle ensures that unauthorized access is blocked by default, reducing the likelihood of accidental or malicious access to sensitive functionality or data. |
| 1. Adhere to the Principle of Least Privilege | Grant users, processes and systems only the permissions they need to perform their tasks. Limiting privileges minimizes the potential damage that could result from a compromise or a coding error. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to external systems or components, sanitize it to ensure it adheres to expected formats and removes malicious content. This prevents vulnerabilities like command injection or cross-site-scripting when interacting with other applications or services. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls to protect against potential threats. If one layer fails, additional layers ensure the system remains secure. For example, combining firewalls, encryption, and authentication adds depth to the defense strategy. |
| 1. Use Effective Quality Assurance Techniques | Regularly test you code using tools and techniques such as static code analysis, dynamic testing, and peer reviews to identify vulnerabilities and ensure adherence to secure coding practices. Quality assurance helps catch issues early and reduces the risks in production. |
| 1. Adopt a Secure Coding Standard | Follow a recognized secure coding standard, such as the SEI CERT C++ Coding Standard, to ensure consistency and security across your codebase. Secure coding standards provide guidelines for avoiding common vulnerabilities and promoting best practices. |

### 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] | This standard involves using strict typing in your code to ensure that there are no runaway generic data types. |

| **Noncompliant Code** |
| --- |
| Using an incorrect or overly broad data type, for example, int instead of unsigned int, which can lead to unexpected behaviors such as storing invalid or negative values. |
| int count = -1  if (count < 0) {  std::cout << “Invalid count!” << std::endl;  } |

| **Compliant Code** |
| --- |
| Using the correct and specific data type, for example, unsigned int, ensures that values are always within valid bounds, preventing unexpected behavior. |
| unsigned int count = 0;  if (count == 0) {  std::cout << “Count is zero.” << std::endl;  } |

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

| **Principles(s):** **Heed Compiler Warnings** – Using strict typing helps catch mismatches early. This standard reinforces the need to resolve compiler warnings to that variables are declared with the most appropriate type. |
| --- |
| **Adopt a Secure Coding Standard** – Enforcing specific data types minimizes the risk of unintended behavior and aligns with best practices for secure, consistent code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Cppcoreguidelines-pro-member-init | Enforces strict initialization of data types to prevent unintended type conversions. |
| SonarQube | 9.9 | Type safety checker | Analyzes code to identify and report on type-related errors. |
| Colverity | 2020.12 | Type Conformance Checker | Automatically detects improper data type usage and potential mismatches. |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | This standard involves checking data to ensure that no unintended out-of-bounds rules are violated. |

| **Noncompliant Code** |
| --- |
| Failing to validate or sanitize the range of values assigned to variables can results in unexpected behaviors or vulnerabilities, such as invalid computations or overflows. |
| int percentage = 150;  std::cout << “Percentage: “ << percentage << std::endl; |

| **Compliant Code** |
| --- |
| Validating or clamping values to the required range ensures predictable and safe program behavior. |
| int percentage = 150;  if (percentage > 100) percentage = 100;  std::cout << “Percentage: “ << percentage << std::endl; |

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

| **Principles(s):** **Validate Input Data** – This standard ensures that values remain within acceptable bounds, protecting against buffer overflows and other errors. |
| --- |
| **Adhere to the Principle of Least Privilege** – Limiting the range of data values helps reduce the risk of unexpected or malicious operations. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.7 | Boundscheck | Detects out-of-bound value errors and ensures data stays within expected ranges. |
| Clang Static Analyzer | 12.0 | Memory Error Checker | Identifies issues related to improper data assignments that could lead to value violations. |
| SonarQube | 9.9 | Range Validation Checker | Checks for data values that exceed defined limits, ensuring compliance with safe range practices. |
|  |  |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | This standard revolves around protecting inputs to ensure no malicious strings are sent to backend services. |

| **Noncompliant Code** |
| --- |
| Failing to validate user input in a string, leading to vulnerabilities like buffer overflow or injecting malicious payloads. |
| char buffer[10];  strcpy(buffer, “This string is way too long for the buffer.”); |

| **Compliant Code** |
| --- |
| Using functions like strncopy or std::string ensures proper bounds checking and precents buffer overflow. |
| char buffer[10];  strncpy(buffer, “Short”, sizeof(buffer) – 1);  buffer[sizeof(buffer) – 1] = ‘\0’; |

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

| **Principles(s):** **Validate Input Data** – Ensures that all string inputs are checked and sanitized before processing, reducing the risk of buffer overflows and injection attacks. |
| --- |
| **Use Effective Quality Assurance Techniques** – Emphasizes rigorous testing and code review to catch errors in string handling that may lead to vulnerabilities. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | String Sanitation Checker | Inspects string operations to ensure proper bounds checking and safe handling of string data. |
| Colverity | 2020.12 | String Handling Checker | Detects potential vulnerabilities in string manipulation, such as buffer overflows and improper termination. |
| SonarQube | 9.9 | String Validation Checker | Analyzes code for unsafe string operations and recommends best practices for secure string handling. |
|  |  |  |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | This standard enforces the requirement of measures against SQL injection, to prevent unwanted manipulation of backend databases. |

| **Noncompliant Code** |
| --- |
| Directly using user input in SQL queries without validation or sanitization makes the application vulnerable to SQL injection attacks. |
| std::string query = “SELECT \* FROM Users WHERE username = ‘” + userInput + “’,’”;  executeQuery(query); |

| **Compliant Code** |
| --- |
| Using parametrized queries ensures that the user input is treated as data, not executable code, thus mitigating SQL injection risks. |
| std::string query = “SELECT \* FROM Users WHERE username = ?”;  prepareAndExecuteQuery(query, userInput); |

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

| **Principles(s):** **Validate Input Data** – Ensures that all user input is rigorously validated to precent malicious data from being injected into SQL queries. |
| --- |
| **Sanitize Data Sent to Other Systems** – Mandates the use of parametrized queries or prepared statements to treat SQL inputs as data rather than executable code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | SQL Injection Checker | Detects concatenated SQL queries and improper handling of user input that could lead to SQL injection vulnerabilities. |
| Colverity | 2020.12 | SQL Injection Vulnerability Checker | Analyzes source code for patterns that may enable SQL injection and recommends the use of parametrized queries. |
| Checkmarx | 8.0 | SQL Injection Scan | Performs deep static analysis to identify SQL injection risks and provides remediation guidance for insecure SQL query construction. |
|  |  |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | This standard enforces the prevention of memory leaks, which can eat up resources if not properly managed. |

| **Noncompliant Code** |
| --- |
| Failing to release dynamically allocated memory can lead to memory leaks, which may degrade system performance over time. |
| int\* ptr = new int[100]; |

| **Compliant Code** |
| --- |
| Using smart pointers, like std::unique\_ptr or std::shared\_ptr ensures that allocated memory is automatically managed and released when no longer needed. |
| #include <memory>  std::unique\_ptr<int[]> ptr = std::make\_unique<int[]>(100); |

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

| **Principles(s):** **Adopt a Secure Coding Standard** – Emphasizes the importance of using best practices, like smart pointers, to prevent memory leaks, ensuring that dynamic memory is managed safely. |
| --- |
| **Practice Defense in Depth** – By ensuring proper memory management, multiple layers of protection are added against resource exhaustion and stability issues. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2020.12 | Memory Leak Checker | Automatically detects memory leaks and improper dynamic memory usage to help prevent resource exhaustion. |
| Clang Static Analyzer | 12.0 | Memory Management Checker | Analyzes code paths for improper memory allocation and deallocation, identifying potential leaks. |
| SonarQube | 9.9 | Memory Safety Checker | Scans for patterns that might lead to memory leaks and improper pointer usage, ensuring resource management best practices. |
|  |  |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are mainly used in development to enforce standards around data to ensure runtime conditions are appropriate. |

| **Noncompliant Code** |
| --- |
| Using assertions is production code to check for user input or runtime conditions is not appropriate as assertions are typically disabled in production builds. |
| int value = -1;  assert(value >= 0); |

| **Compliant Code** |
| --- |
| Assertions should only be used to enforce programming assumptions during development, not for runtime checks. For runtime conditions, use explicit error handling. |
| Int value = -1;  Assert(value >= 0 && “Value must be non-negative”); |

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

| **Principles(s):** **Heed Compiler Warnings** – Using assertions correctly in development helps identify problematic code early and reinforces the need to address warnings before production. |
| --- |
| **Use Effective Quality Assurance Techniques** – Assertions during development act as an internal check to ensure that code assumptions hold, thus contributing to robust testing practices. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Assertion Usage Checker | Reviews code to ensure assertions are used appropriately during development and are not relied upon in production. |
| Cppcheck | 2.7 | Assertion Analysis Checker | Analyzes code to identify misuse of assertions and ensures that they are confined to development builds. |
| SonarQube | 9.9 | Code Quality Checker | Flags inappropriate usage of assertions in production code and recommends better error handling approaches. |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | This standard enforces the catching of certain exceptions in order to properly notify the server or user instead of breaking functionality. |

| **Noncompliant Code** |
| --- |
| Catching general exceptions with a catch statement without specific handling can obscure the root cause of errors and make debugging difficult. |
| try {  int\* array = new int[10];  throw std::runtime\_error("An error occurred");  } catch (...) {  std::cout << "An error occurred." << std::endl;  } |

| **Compliant Code** |
| --- |
| Catching specific exceptions and providing detailed error handling makes the code more maintainable and easier to debug. |
| try {  int\* array = new int[10];  throw std::runtime\_error("An error occurred");  } catch (const std::runtime\_error& e) {  std::cout << "Runtime error: " << e.what() << std::endl;  } catch (const std::exception& e) {  std::cout << "Exception: " << e.what() << std::endl;  } |

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

| **Principles(s):** **Adopt a Secure Coding Standard** – By catching specific exceptions, the code follows best practices that promote clear error handling and prevent the masking of issues. |
| --- |
| **Practice Defense in Depth** – Proper exception handling adds an extra layer of protection by ensuring that errors are managed gracefully without exposing sensitive information or crashing the application. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2020.12 | Exception Handling Checker | Automatically detects overly broad exception handlers and recommends catching specific exceptions. |
| Clang Static Analyzer | 12.0 | Exception Safety Checker | Analyzes code to ensure that exception handling practices follow secure coding guidelines and do not obscure errors. |
| SonarQube | 9.9 | Exception Handling Analyzer | Flags improper exception catching that might lead to unhandled cases or masked errors, guiding developers toward more robust error management. |
|  |  |  |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | [STD-008-CPP] | This standard enforces proper handling of file success/failure responses to prevent unintended results. |

| **Noncompliant Code** |
| --- |
| Failing to check the success of file operations can lead to undefined behavior or silent failures. |
| std::ofstream file(“data.txt”);  file << “Important data”;  file.close(); |

| **Compliant Code** |
| --- |
| Always check that a file operation succeeded before attempting to use the file. Use RAII techniques when possible for automatic cleanup. |
| #include <iostream>  #include <fstream>  std::ofstream file("data.txt");  if (!file.is\_open()) {  std::cerr << "Failed to open file!" << std::endl;  } else {  file << "Important data";  } |

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

| **Principles(s):** **Validate Input Data** – Ensuring that file operations are verified reinforces the need to check that external interactions (such as file reads/writes) are safe and valid. |
| --- |
| **Use Effective Quality Assurance Techniques** – Proper file handling, including checking for success or failure, is part of rigorous testing and quality control to prevent unexpected behavior. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | I/O Checker | Scans code for file handling practices to ensure that file operations are properly checked for success and error handling is in place. |
| Coverity | 2020.12 | File Operation Validator | Detects potential issues in file open/read/write operations and ensures that errors are appropriately handled. |
| Clang Static Analyzer | 12.0 | Resource Handling Checker | Reviews code to verify that file streams are correctly manages, including error checking and resource cleanup. |
|  |  |  |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging Coding | [STD-009-CPP] | This standard enforces non-sensitive logging for functionalities that require operations to be logged to the console. |

| **Noncompliant Code** |
| --- |
| Failing to sanitize sensitive data before logging can expose confidential information, leading to security breaches. |
| std::string password = "SuperSecret123";  std::cout << "Password: " << password << std::endl; |

| **Compliant Code** |
| --- |
| Avoid logging sensitive data such as passwords, tokens, or personally identifiable information. |
| std::string password = "SuperSecret123";  std::cout << "Password updated successfully." << std::endl; |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems** – Ensures that logs do not contain sensitive information by mandating proper sanitization of data before logging, reducing the risk of data leaks. |
| --- |
| **Use Effective Quality Assurance Techniques** – Emphasizes that logging should be implemented thoughtfully to capture necessary operational data without compromising security or performance. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Logging Sanitation Checker | Scans for instances where sensitive data might be inadvertently logged and recommends remediation. |
| Coverity | 2020.12 | Sensitive Data Logging Checker | Analyzes logging code to ensure that confidential information is not being output in logs. |
| Clang Static Analyzer | 12.0 | Log Verification Checker | Review code for proper logging practices and verifies that only non-sensitive information is captured. |
|  |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | [STD-010-CPP] | This standard enforces safe use of pointers to avoid common pitfalls like segmentation faults. |

| **Noncompliant Code** |
| --- |
| Using raw pointers without proper initialization or cleanup can result in unexpected behavior. |
| int\* ptr;  \*ptr = 10; |

| **Compliant Code** |
| --- |
| Always initialize pointers, use smart pointers where applicable, and avoid dangling pointers by ensuring proper cleanup. |
| #include <memory>  std::unique\_ptr<int> ptr = std::make\_unique<int>(10);  std::cout << \*ptr << std::endl; |

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

| **Principles(s):** **Heed Compiler Warnings** – Emphasizes the importance of addressing compiler warnings related to pointer usage, ensuring that pointer-related errors are caught early. |
| --- |
| **Adopt a Secure Coding Standard** – Enforces best practices for pointer management (such as using smart pointers) to avoid common pitfalls like segmentation faults and memory leaks. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | High | Critical | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Pointer Safety Checker | Inspects code for unsafe pointer operations and suggests improvements such as replacing raw pointers with smart pointers. |
| Coverity | 2020.12 | Dangling Pointer Detector | Automatically detects improper pointer usage, including uninitialized or dangling pointers, to prevent segmentation faults. |
| SonarQube | 9.9 | Memory Management Checker | Analyzes pointer usage patterns and flags potential issues in pointer arithmetic or misuse that could lead to crashes or undefined behavior. |
|  |  |  |  |

### 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 | Medium | Medium | High | 3 |
| STD-003-CPP | High | Medium | Medium | High | 3 |
| STD-004-CPP | Critical | High | Medium | Critical | 4 |
| STD-005-CPP | High | Medium | Medium | High | 3 |
| STD-006-CPP | Medium | Medium | Low | Medium | 2 |
| STD-007-CPP | High | Medium | Low | High | 3 |
| STD-008-CPP | Medium | High | Low | Medium | 3 |
| STD-009-CPP | Medium | Medium | Low | Medium | 2 |
| STD-010-CPP | Critical | High | High | Critical | 4 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 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 | Encryption at rest protects data stored on physical media (hard drives, SSDs, cloud storage, etc.) by converting it into a secure format that is unreadable without the appropriate decryption key. This policy applies to any stored sensitive data to mitigate the risk of data breaches resulting from unauthorized physical or network access. By ensuring that data remains encrypted when not in active use, even if storage devices are compromised, the organization can reduce the impact of data theft. |
| Encryption in flight | Encryption in flight (or in transit) secures data as it moves across networks using protocols such as TLS/SSL. This policy ensures that data sent between clients, servers, or across distributed systems is protected from interception or tampering during transmission. It is especially critical when transmitting sensitive information over public or untrusted networks. |
| Encryption in use | Encryption in use refers to methods that protect data while it is being processed. Emerging technologies such as homomorphic encryption or secure enclaves allow computations to be performed on encrypted data. This policy minimizes exposure of sensitive data during processing, ensuring that even if an application or system is compromised, the data being used is not readily accessible in plaintext. This is crucial for high-security environments handling confidential or regulated information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user or system. The policy mandates robust authentication mechanisms (e.g., multi-factor authentication, secure password policies) to ensure that only authorized individuals can access systems. This is vital for protecting user logins and preventing unauthorized access. |
| Authorization | Authorization determines the level of access and permissions an authenticated user has within a system. By enforcing strict authorization policies, the organization ensures that users can perform only those actions that are essential for their role. This controls changes to databases, the addition of new users, and restricts access to sensitive files or system functions. |
| Accounting | Accounting involves tracking and recording user activities, changes, and access within the system. This policy requires comprehensive logging and audit trails for all key actions (e.g., file accesses, modifications, logins, and changes to user privileges). It ensures traceability, supports incident investigations, and helps maintain accountability across the system. |

**\***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.

**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 |  |
| 2.0 | 01/26/2025 | Added security principles and their definitions. | John Schatzl |  |
| 3.0 | 02/16/2025 | Added applicable rules, risk assessments, automation tools, and policies. | John Schatzl |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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