**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 | Every piece of data provided by users or external systems should be validated before use. Validating input helps protect against common vulnerabilities such as SQL injection, XSS, and buffer overflows. By ensuring that all inputs conform to expected formats and contain no harmful content, systems can prevent many types of attacks that exploit input data. |
| 1. Heed Compiler Warnings | Compiler warnings are often indicators of potential security flaws or bugs in the code. By paying attention to and addressing these warnings, developers can prevent security issues from reaching production. Resolving all warnings helps enhance the overall security and stability of applications. |
| 1. Architect and Design for Security Policies | Systems should be designed and architected with security as a fundamental principle. This involves defining and integrating security policies during the design phase, which helps in creating a robust system architecture resistant to attacks. This principle emphasizes the need for security to be considered from the start, rather than being added as an afterthought. |
| 1. Keep It Simple | Complexity is the enemy of security. A simple, clean, and understandable design is easier to audit and less likely to contain security bugs. Keeping software designs as simple as possible helps in maintaining, scaling, and securing them more effectively. |
| 1. Default Deny | Systems should deny all access by default, only permitting interactions that have been explicitly allowed. This security model ensures that only authorized actions are permitted, minimizing the risk of unauthorized access and reducing the attack surface. |
| 1. Adhere to the Principle of Least Privilege | This principle involves limiting access rights for users, accounts, and computing processes to only those resources absolutely required to perform their tasks. By minimizing user and process privileges, the damage potential from an exploit or error is greatly reduced. |
| 1. Sanitize Data Sent to Other Systems | Data that is transmitted to external systems must be sanitized to prevent the unintentional leakage of sensitive information or the execution of malicious code. Ensuring that outgoing data is cleansed of harmful content protects both the sending system and the receiving system. |
| 1. Practice Defense in Depth | Implementing multiple layers of security measures to protect data and operations. This approach does not rely on a single security mechanism but instead uses a series of overlapping and redundant defensive measures to protect information even if one layer fails. |
| 1. Use Effective Quality Assurance Techniques | Applying rigorous and effective quality assurance practices helps identify and mitigate risks before software is deployed. These techniques include thorough testing, code reviews, and the use of automated tools to detect and resolve security vulnerabilities early in the development lifecycle. |
| 1. Adopt a Secure Coding Standard | Using a well-defined coding standard that emphasizes security can guide developers towards writing safer code. Such standards help prevent common coding errors that could lead to security vulnerabilities by providing guidelines and best practices for secure software development. |

### 

### 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 | Strong type safety prevents type mismatches that could lead to security vulnerabilities like buffer overflows and type confusion. |

| **Noncompliant Code** |
| --- |
| Implicit type conversion can lead to data loss, which might cause logic errors and vulnerabilities. |
| int index;  double pi = 3.14159;  index = pi; // Implicit conversion |

| **Compliant Code** |
| --- |
| Using explicit casting prevents unintended data conversions, maintaining integrity and preventing errors. |
| int index;  double pi = 3.14159;  index = static\_cast<int>(pi); // Explicit conversion |

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

| **Principles(s):** Adhere to the Principle of Least Privilege. Strong type safety enforces correct type usage and limits the operations that can be performed with each type, effectively minimizing privileges and reducing the likelihood of type-related security exploits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | C++ Code Quality and Security | Analyzes C++ code to detect type mismatches, unsafe type conversions, and potential type-related vulnerabilities, enhancing both security and stability. |
| Clang Static Analyzer | 14.0 | Type Safety Check | Uses static analysis to identify potential type safety violations in C++ code, aiming to prevent bugs and security issues related to improper type usage. |
| ReSharper C++ | 2022.3 | Type Safety Inspections | A Visual Studio extension that inspects C++ code for type safety issues and suggests corrections, helping maintain robust and error-free code. |
| Visual Studio Code Analysis | 2023 | C++ Core Guidelines Checker | Provides built-in support for enforcing C++ Core Guidelines, including strong type safety practices, directly within the Visual Studio development environment. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Evaluating expressions at compile time reduces runtime overhead and potential runtime errors from unexpected values. |

| **Noncompliant Code** |
| --- |
| Calculating power at runtime can introduce performance overhead and potential calculation errors. |
| const int base = 10;  int exp = 3;  int result = pow(base, exp); // Evaluated at runtime |

| **Compliant Code** |
| --- |
| Using constexpr allows the computation to be done at compile time, ensuring correctness and efficiency. |
| constexpr int base = 10;  constexpr int exp = 3;  constexpr int result = pow(base, exp); // Evaluated at compile-time |

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

| **Principles(s):** Keep It Simple: Using constant expressions simplifies the code by resolving values at compile time, avoiding runtime complications and reducing possible points of failure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC | 11.2 | Compile-time Evaluation | GCC evaluates constant expressions at compile time, optimizing runtime performance and ensuring that constants are handled correctly. |
| Clang | 14.0 | Expression Evaluator | Provides diagnostics and optimizations for constant expressions in C++ code, ensuring efficient and error-free compile-time calculations. |
| MSVC | 2023 | Constant Expression Evaluation | Microsoft's Visual C++ compiler checks for expressions that can be evaluated as constexpr, promoting efficient code by verifying and enforcing compile-time evaluations. |
| Intel C++ Compiler | 2022 | Constant Evaluation Check | Offers optimizations and checks for constant evaluations in C++ code, improving both the security and performance of applications. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Proper handling of strings is crucial for avoiding buffer overflows and ensuring data integrity. |

| **Noncompliant Code** |
| --- |
| This code causes a buffer overflow by copying a string longer than the buffer. |
| char buf[10];  strcpy(buf, "overlong string input"); // Buffer overflow |

| **Compliant Code** |
| --- |
| Using strncpy and ensuring null-termination prevents buffer overflow. |
| char buf[10];  strncpy(buf, "input", sizeof(buf) - 1);  buf[sizeof(buf) - 1] = '\0'; // Null-termination |

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

| **Principles(s):** Validation Input Data: Proper string handling ensures that all inputs are correctly bounded and treated, which directly supports the validation of input data to prevent common issues like buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.12 | String Manipulation Errors | Identifies potential string handling errors such as buffer overflows, out-of-bounds errors, and improper null termination in C++ code. |
| Fortify Static Code Analyzer | 22.2 | Buffer Overflow Detection | Analyzes C++ code to detect buffer overflows and other memory corruption vulnerabilities, offering suggestions to improve string safety. |
| Klocwork | 2023 | String Safety Analysis | Provides in-depth analysis of string usage in C++ code, detecting potential security risks and suggesting safer alternatives. |
| CodeQL | 2.8 | Unsafe String Operations | Uses query-based code analysis to find vulnerabilities related to unsafe string operations, enhancing code security through deep semantic analysis. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL injection is essential for safeguarding database integrity and security. |

| **Noncompliant Code** |
| --- |
| This example shows how a SQL command can be maliciously altered through user input. |
| std::string user\_input = "'; DROP TABLE users; --";  std::string sql = "SELECT \* FROM users WHERE name = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| Parameterized queries prevent SQL injection by separating SQL logic from data. |
| std::string user\_input = getUserInput();  std::string sql = "SELECT \* FROM users WHERE name = ?"; // Use parameterized queries |

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

| **Principles(s):** Sanitize Data Sent to Other Systems: SQL injection prevention is a direct application of sanitizing data, ensuring that all database queries are safe from injection attacks by separating data from command execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | SQL Injection Detection | Analyzes potential SQL injection vulnerabilities in C++ code, providing fixes and recommendations for secure coding practices. |
| Fortify Static Code Analyzer | 22.2 | SQL Injection | Scans for SQL injection patterns and offers detailed vulnerability management and mitigation recommendations. |
| Checkmarx | 2023 | SQL Injection | Performs static code analysis to detect SQL injection vulnerabilities and educates developers on secure coding practices. |
| Veracode | 10.12 | SQL Injection Flaws | Offers application security testing and static analysis, identifying SQL injection risks and providing solutions. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Safe memory operations are crucial to prevent buffer overflows and unauthorized memory access. |

| **Noncompliant Code** |
| --- |
| This example causes a buffer overflow due to unsafe string copying. |
| char \*buf = new char[10];  strcpy(buf, "This is definitely too long!"); |

| **Compliant Code** |
| --- |
| Using snprintf ensures that the buffer boundaries are respected, preventing overflows. |
| char \*buf = new char[10];  snprintf(buf, 10, "%s", "short"); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques: Ensures memory protection practices are properly implemented to prevent defects that could compromise system integrity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18.1 | Memory Errors | Memory debugging tool that detects memory leaks, overflows, and other memory-related errors. |
| Address Sanitizer | LLVM 14.0 | Memory Corruption | A runtime memory error detector designed for fast execution and detailed error reporting of memory misuse. |
| Dr. Memory | 2.3.0 | Memory Diagnosis | Efficiently diagnoses and reports memory-related errors like leaks and corruption in complex software. |
| IBM Rational PurifyPlus | Latest | Memory Access Errors | Detects and helps debug memory allocation errors and threading errors in real-time environments. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions help catch bugs by verifying assumptions during development but should not be used for handling runtime errors. |

| **Noncompliant Code** |
| --- |
| Assertions are disabled in release builds, leaving potential for division by zero in production. |
| int divide(int x, int y) {  assert(y != 0);  return x / y;  } |

| **Compliant Code** |
| --- |
| Proper error handling with exceptions prevents division by zero in all builds. |
| int divide(int x, int y) {  if (y == 0) throw std::invalid\_argument("divisor cannot be zero");  return x / y;  } |

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

| **Principles(s):** Keep It Simple: Proper use of assertions simplifies debugging by clearly indicating programmer expectations and error conditions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPP Check | 2.8 | Misused Assertions | Static analysis tool that checks for misuse or overuse of assertions in C++ code. |
| Clang Static Analyzer | 14.0 | Assertion Usage | Analyzes use of assertions in C++ to ensure they are used correctly and effectively. |
| Visual Studio Code Analysis | 2023 | Assertion Analysis | Provides in-depth analysis of assertions within the Visual Studio IDE to ensure they adhere to best practices. |
| PVS-Studio | 7.21 | General Analysis including Assertions | Checks C++ code for potential errors and misuse of assertions, suggesting better practices. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper use of exceptions ensures that errors are handled gracefully and resources are freed appropriately. |

| **Noncompliant Code** |
| --- |
| Returning nullptr on error can lead to dereferencing null pointers. |
| int \*allocateArray(int size) {  int \*arr = new int[size];  if (size < 0) return nullptr; // Improper error handling  return arr;  } |

| **Compliant Code** |
| --- |
| Throwing an exception for invalid input ensures issues are handled before memory allocation. |
| int \*allocateArray(int size) {  if (size < 0) throw std::invalid\_argument("size must be non-negative");  int \*arr = new int[size];  return arr;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard**:** Proper exception handling is part of secure coding standards that ensure robust and error-resilient applications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | Exception Handling | Ensures exceptions are used properly and handled correctly across different layers of the application. |
| ReSharper C++ | 2022.3 | Exception Usage Inspection | Helps identify and correct improper use of exceptions, improving code reliability and maintainability. |
| CodeQL | 2.8 | Exception Usage Inspection | Uses query-based analysis to identify exceptions that could lead to resource leaks or inconsistent application states. |
| Klocwork | 2023 | Exception Handling Analysis | Detects potential issues in exception handling practices, ensuring that exceptions do not lead to resource leaks or crashes. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-008-CPP | RAII (Resource Acquisition Is Initialization) ensures that resources such as memory, file handles, and network connections are properly released. |

| **Noncompliant Code** |
| --- |
| This code does not close the file handle if an error occurs after opening. |
| FILE \*file = fopen("data.txt", "r");  if (!file) return; // File not closed on error path  fclose(file); |

| **Compliant Code** |
| --- |
| Using C++ streams ensures that file handles are closed automatically, even if an exception is thrown. |
| std::ifstream file("data.txt");  if (!file) throw std::runtime\_error("Failed to open file");  // File is automatically closed when `file` goes out of scope |

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

| **Principles(s):** Adopt a Secure Coding Standard: RAII is a fundamental secure coding practice in C++ to manage resource lifecycles, ensuring that all resources are properly released to avoid leaks and undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18.1 | Resource Leak Detection | Identifies memory leaks and other resource mismanagement issues, crucial for RAII enforcement. |
| CPP Check | 2.8 | Resource Management | Static analysis tool that checks for potential resource management errors, including incorrect RAII patterns. |
| Clang-Tidy | 14.0 | Modernize Use RAII | Automatically checks for and applies modern C++ practices, including RAII, to manage resources more safely and efficiently. |
| Visual Leak Detector | 2.5.1 | Windows Resource Leak Detection | Specifically designed for detecting memory leaks in Windows applications using C++. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | STD-009-CPP | Ensuring thread safety in concurrent executions to prevent race conditions and deadlocks. |

| **Noncompliant Code** |
| --- |
| Access to the shared variable without synchronization can lead to race conditions. |
| int shared = 0;  std::thread t1([&]() { shared = 1; });  std::thread t2([&]() { shared = 2; });  t1.join();  t2.join(); |

| **Compliant Code** |
| --- |
| Using std::atomic for shared variables ensures atomic operations and prevents race conditions. |
| std::atomic<int> shared = 0;  std::thread t1([&]() { shared.store(1); });  std::thread t2([&]() { shared.store(2); });  t1.join();  t2.join(); |

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

| **Principles(s):** Practice Defense in Depth: Ensuring thread safety in concurrent executions protects against race conditions and deadlocks, crucial for maintaining application integrity and reliability in multi-threaded environments. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Intel Inspector | 2023 | Threading Error Analysis | Identifies race conditions and deadlocks in multi-threading code, ensuring thread safety. |
| Helgrind | 3.18.1 | Race Condition Detector | Detects synchronization errors and potential race conditions in C++ programs. |
| ThreadSanitizer | LLVM 14.0 | Data Race Detector | A runtime tool that detects data races in C++ applications, supporting better concurrency handling. |
| Microsoft Concurrency Visualizer | 2023 | Concurrency and Threading Efficiency | Part of Visual Studio that visualizes threading and synchronization information, helping to optimize and debug multi-threaded applications. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-010-CPP | Comprehensive error handling ensures that errors are detected early and managed consistently across the application. |

| **Noncompliant Code** |
| --- |
| Using magic numbers for error indication is error-prone and can lead to unclear code. |
| int process(int x) {  if (x < 0) return -1; // Error indicated by special return value  return x + 1;  } |

| **Compliant Code** |
| --- |
| Using exceptions for error handling clarifies the error paths and ensures that they are handled explicitly. |
| int process(int x) {  if (x < 0) throw std::domain\_error("Negative input not allowed");  return x + 1;  } |

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

| **Principles(s):** Keep It Simple and Adopt a Secure Coding Standard: Comprehensive error handling simplifies debugging and maintenance, ensures consistent application behavior, and enhances security by preventing error-related vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | C++ Error Handling | Analyzes error handling patterns to ensure robust and clear error handling strategies are employed. |
| ReSharper C++ | 2022.3 | Exception Usage Inspection | Provides analysis and suggestions to improve exception handling and error reporting in C++ code. |
| CPP Check | 2.8 | Exception Safety | Checks for potential exceptions safety issues and suggests how to handle C++ exceptions more securely. |
| Coverity | 2022.12 | Error Handling Issues | Static analysis tool that identifies issues in error handling logic, providing recommendations to improve safety and compliance. |

### 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 | Medium | Medium | Low | High | 3 |
| STD-002-CPP | Low | Low | Low | Medium | 2 |
| STD-003-CPP | High | High | Medium | High | 5 |
| STD-004-CPP | High | Medium | High | High | 4 |
| STD-005-CPP | High | Medium | High | High | 4 |
| STD-006-CPP | Medium | Low | Medium | Medium | 3 |
| STD-007-CPP | High | Medium | Medium | High | 4 |
| STD-008-CPP | High | Medium | Medium | High | 4 |
| STD-009-CPP | High | High | High | High | 5 |
| STD-0010-CPP | High | Medium | Medium | High | 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 refers to encrypting data that is stored on a physical medium, such as hard drives, SSDs, or external storage devices. This type of encryption is crucial for protecting data from being accessed by unauthorized individuals who might gain physical access to the storage media.  Policy Application: The policy should require that all sensitive or personally identifiable information (PII) stored in databases, file systems, and other storage solutions be encrypted using strong encryption standards such as AES (Advanced Encryption Standard) 256-bit. This ensures that even if the physical security measures fail or storage devices are stolen, the data remains protected and inaccessible without the proper decryption keys. |
| Encryption in flight | Encryption in flight involves encrypting data while it is being transmitted across a network. This prevents unauthorized interception and access during transmission, protecting the data from eavesdropping, interception, and manipulation.  Policy Application: The policy mandates that all data transmitted over public or untrusted networks must be encrypted using secure protocols such as TLS (Transport Layer Security) or VPNs (Virtual Private Networks). This applies to data exchanges between internal systems, as well as communications between the company's systems and external parties. Enforcing encryption in flight protects data integrity and privacy by ensuring that data cannot be read or tampered with during transmission. |
| Encryption in use | Encryption in use refers to techniques that protect data that is actively being processed or accessed. This type of encryption ensures that data remains secure even when it is loaded into memory and being used by applications.  Policy Application: The policy should specify that any operational processes involving sensitive data, such as processing in cloud environments or other shared infrastructure, must employ mechanisms to keep the data encrypted while in use. Technologies like Homomorphic Encryption or Trusted Execution Environments (TEEs) can be utilized. This ensures that data remains protected not only at rest and in transit but also when it is most vulnerable to insider threats and memory attacks.  By implementing and adhering to these encryption standards as outlined in the security policy, organizations can significantly enhance their data security posture and reduce the risk of data breaches and leaks across all stages of data handling. |

| 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, device, or other entity in a computer system, often as a prerequisite to allowing access to resources in a system.  Policy Application: The policy should require users to prove their identity before accessing any systems or data. This is typically achieved through mechanisms like passwords, biometrics, security tokens, or multi-factor authentication (MFA). The policy must ensure that authentication mechanisms are strong enough to withstand attacks and that they are implemented consistently across all systems. |
| Authorization | Authorization involves granting or denying rights to resources based on an authenticated identity's permissions. It determines what users can and cannot do within a system, based on their roles or attributes.  Policy Application: After authentication, the policy should define what resources a user can access and what actions they can perform. This should be managed through role-based access control (RBAC) or attribute-based access control (ABAC), ensuring users receive access rights consistent with their responsibilities in the organization. |
| Accounting | Accounting is the practice of tracking user activities and recording security-relevant information. It is used to provide evidence of compliance with policies and to help detect and respond to security incidents.  Policy Application: The policy should ensure that detailed logs of user activities are maintained, including successful and failed access attempts, changes made during sessions, and system-level events. This enables the organization to detect anomalies, support forensic activities, and ensure compliance with regulatory requirements. |

**\***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 | 01/24/2025 | 3-2 Milestone | Gavin Bish |  |
| 3.0 | 02/11/2025 | Project One | Gavin Bish |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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