**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 should be validated to prevent malicious code injection, buffer overflows, and unexpected crashes. This ensures only properly formatted and expected data is processed. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential vulnerabilities such as buffer overflows or type mismatches. Developers should resolve all warnings to ensure code reliability and security. |
| 1. Architect and Design for Security Policies | Security should be a fundamental part of the software design process. This includes defining clear policies for authentication, authorization, and secure data handling. |
| 1. Keep It Simple | Simpler code is easier to audit and less likely to have hidden security flaws. Avoiding overly complex structures reduces the risk of unintended behavior. |
| 1. Default Deny | Access control should be based on the principle of default deny, meaning that users and processes are granted only the minimum permissions required for their tasks. |
| 1. Adhere to the Principle of Least Privilege | Users and applications should run with the minimum necessary privileges to reduce the potential impact of security breaches. |
| 1. Sanitize Data Sent to Other Systems | All data sent to external systems should be sanitized to prevent injection attacks and corruption, ensuring safe interoperability. |
| 1. Practice Defense in Depth | Multiple layers of security mechanisms should be implemented to reduce the risk of a single point of failure. |
| 1. Use Effective Quality Assurance Techniques | Regular code reviews, automated testing, and static analysis tools should be used to identify and fix vulnerabilities before deployment. |
| 1. Adopt a Secure Coding Standard | Following industry best practices, such as the SEI CERT C++ Coding Standard, helps ensure that software is robust against common security threats. |

### 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-C++ | Use Explicit Data Types |

| **Noncompliant Code** |
| --- |
| The implicit conversion may result in data loss. |
| int x = 3.5; // Implicit conversion from double to int |

| **Compliant Code** |
| --- |
| Explicit data types prevent unintended truncation. |
| double x = 3.5; // Explicitly declares the variable as a double |

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

| **Principles(s):**  **1. Validate Input Data** – This principle ensures only expected and properly formatted data is processed. Using explicit data types enforces this expectation and avoids unintended type coercion or conversion. **2. Heed Compiler Warnings** – Implicit conversions typically generate compiler warnings. Addressing these by using explicit data types helps resolve potential vulnerabilities at compile time. **10. Adopt a Secure Coding Standard** – Following this standard is consistent with SEI CERT recommendations for strong typing to prevent unintended behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | implicitConversion | Flags implicit conversions that may cause truncation or loss of data. |
| SonarQube | 10.2 | cpp:S1939 | Detects dangerous implicit conversions. |
| Clang-Tidy | 17.0 | clang-analyzer-core.CallAndMessage | Highlights implicit conversion and type safety issues. |
| Visual Studio Analyzer | 2022 | Compiler Warning C4244 | Warns about data loss due to type conversion. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-C++ | Avoid Hardcoded Values |

| **Noncompliant Code** |
| --- |
| Hardcoded values make maintenance difficult and can introduce security risks if sensitive information is exposed. |
| int maxUsers = 1000; |

| **Compliant Code** |
| --- |
| Using named constants instead of hardcoded values improves code maintainability, readability, and security by preventing unintended modifications. |
| const int MAX\_USERS = 1000; |

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

| **Principles(s):**  **4. Keep It Simple** – Defining constants in a centralized and consistent way improves code clarity and prevents hidden logic that may confuse future developers. **6. Adhere to the Principle of Least Privilege** – Avoiding hardcoded access credentials or configuration values minimizes accidental over-permissioning or exposure. **10. Adopt a Secure Coding Standard** – Using constants in place of magic numbers aligns with best practices in secure and maintainable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | magicNumber | Detects magic numbers and recommends defining constants. |
| SonarQube | 10.2 | cpp:S109 | Flags hardcoded values and recommends using named constants. |
| Clang-Tidy | 17.0 | modernize-use-const | Suggests making immutable values const and using named identifiers. |
| Visual Studio Analyzer | 2022 | Manual Inspection | No specific warning for magic numbers—requires code review for context. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-C++ | Use Safe String Functions |

| **Noncompliant Code** |
| --- |
| Using unsafe string functions like strcpy can cause buffer overflows, leading to security vulnerabilities. |
| char buffer[10];  strcpy(buffer, "This is too long!"); |

| **Compliant Code** |
| --- |
| Using strncpy ensures that the buffer does not exceed its allocated size, preventing buffer overflows. |
| 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):**  **1. Validate Input Data** – This standard protects against unsafe input operations by using bounded functions to limit data sizes. **8. Practice Defense in Depth** – Prevents overflow vulnerabilities, which are common attack vectors in layered exploitation. **10. Adopt a Secure Coding Standard** – Encourages the use of safe and modern functions over legacy, unsafe methods. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | [Insert text.] | unsafeFunctionUsage | Detects usage of unsafe string functions like strcpy, strcat, etc. |
| SonarQube | [Insert text.] | cpp:S586 | Flags unsafe use of string functions and suggests safer alternatives. |
| Clang-Tidy | [Insert text.] | cert-err34-c | Identifies dangerous string handling practices. |
| Fortify SCA | [Insert text.] | Buffer Overflow: String | Detects potential buffer overflows due to unsafe string usage. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-C++ | Use Prepared Statements |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries makes the application vulnerable to SQL injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "';"; |

| **Compliant Code** |
| --- |
| Using prepared statements ensures that user input is properly parameterized, preventing SQL injection. |
| std::unique\_ptr<sql::PreparedStatement> stmt(conn->prepareStatement("SELECT \* FROM users WHERE username = ?"));  stmt->setString(1, username); |

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

| **Principles(s):**  **3. Architect and Design for Security Policies** – Enforcing proper input handling in the architecture prevents malicious injection. **5. Default Deny** – SQL prepared statements restrict what input is allowed and interpreted, reducing injection opportunities. **7. Sanitize Data Sent to Other Systems** – This standard prevents malicious data from being injected into backend SQL systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | [Insert text.] | sql-injection | Detects raw SQL strings that include user input and recommends prepared statements. |
| Fortify SCA | [Insert text.] | SQL Injection | Analyzes data flow for unsafe SQL construction using user input. |
| Cppcheck | [Insert text.] | (Custom rules required) | Can be extended to detect SQL usage patterns with user input. |
| Checkmarx | [Insert text.] | Code Injection - SQL | Detects all types of injection vulnerabilities, including improper SQL usage. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-C++ | Proper Memory Management |

| **Noncompliant Code** |
| --- |
| Failing to free dynamically allocated memory can lead to memory leaks, reducing system performance over time. |
| int\* ptr = new int[10];  // No delete operation, causing a memory leak |

| **Compliant Code** |
| --- |
| Using delete[] ensures that allocated memory is properly deallocated, preventing memory leaks. Alternatively, using std::unique\_ptr automates memory management. |
| int\* ptr = new int[10];  delete[] ptr; // Properly deallocates memory |

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

| **Principles(s):**  **2. Heed Compiler Warnings** – Compilers often flag memory leaks or usage of unfreed memory, helping catch issues early. **4. Keep It Simple** – Using smart pointers simplifies memory management and reduces human error. **9. Use Effective Quality Assurance Techniques** – Static analysis tools can detect leaks, and automated tests can catch improper memory handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | Memory Leak Check | Runtime analysis tool that detects memory leaks, invalid reads/writes, and more. |
| AddressSanitizer | GCC 12+ / Clang 16+ | N/A | Runtime sanitizer that finds memory errors such as buffer overflows and leaks. |
| Clang-Tidy | 15.0 | clang-analyzer-cplusplus.NewDeleteLeaks | Checks for memory that was allocated but not deallocated. |
| Cppcheck | 2.13 | memleak | Detects memory leaks caused by missing deallocations. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-C++ | Use Assertions for Debugging |

| **Noncompliant Code** |
| --- |
| Not using assertions can lead to undetected logic errors, potentially causing undefined behavior at runtime. |
| int divide(int x, int y) {  return x / y; // No validation for division by zero  } |

| **Compliant Code** |
| --- |
| Using assertions ensures that invalid conditions, such as division by zero, are caught during development, preventing runtime errors. |
| #include <cassert>  int divide(int x, int y) {  assert(y != 0 && "Division by zero error!"); // Ensures valid input  return x / y;  } |

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

| **Principles(s):**  **2. Heed Compiler Warnings** – Some compilers may issue warnings about undefined behavior; assertions make intent explicit. **9. Use Effective Quality Assurance Techniques** – Assertions are valuable in unit testing and catching logic flaws early during development. **10. Adopt a Secure Coding Standard** – Assertions are a recommended part of development in many coding guidelines, including SEI CERT. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | bugprone-assert-side-effect | Detects assertions with side effects that may not be executed in release builds. |
| Cppcheck | 2.13 | assertWithSideEffect | Warns about assertions that contain side effects or risky conditions. |
| Visual Studio | 2022 | Compiler Warnings | Can emit warnings when division by zero is statically possible. |
| SonarQube | 10.3 | c:LogicError | Identifies logic errors such as missing assertions or validation in critical paths. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-C++ | Handle Exceptions Properly |

| **Noncompliant Code** |
| --- |
| Catching all exceptions with a generic catch block (catch (...)) without handling the error can obscure bugs and make debugging difficult. |
| try {  riskyFunction();  } catch (...) {  // Silent catch - error is not handled  } |

| **Compliant Code** |
| --- |
| Catching specific exceptions allows for proper error handling, ensuring that issues are logged and addressed appropriately. |
| #include <iostream>  #include <exception>  try {  riskyFunction();  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what() << std::endl;  } catch (const std::exception& e) {  std::cerr << "Exception: " << e.what() << std::endl;  } |

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

| **Principles(s):**  **3. Architect and Design for Security Policies** – Ensures that error handling is built into the software architecture to address failures gracefully. **9. Use Effective Quality Assurance Techniques** – By catching specific exceptions, developers can more easily identify bugs and write tests that validate error handling. **4. Keep It Simple** – Targeted exception handling is clearer and easier to debug than generic catch-all blocks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-err60-cpp | Flags use of overly generic exception handling or lack of error reporting. |
| Cppcheck | 2.13 | catchExceptionByValue | Warns when exceptions are not caught by reference or when catch(...) is overused. |
| SonarQube | 10.3 | cpp:S131 | Ensures exceptions are not silently caught without proper handling. |
| Visual Studio | 2022 | Code Analysis | Can detect poor exception handling practices with appropriate rule sets. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-008-C++ | Validate User Input |

| **Noncompliant Code** |
| --- |
| Accepting user input without validation can lead to unexpected behavior and security vulnerabilities, such as buffer overflows or injection attacks. |
| char username[20];  std::cin >> username; // No input validation |

| **Compliant Code** |
| --- |
| Using input validation ensures that only expected data is processed, preventing security risks. |
| #include <iostream>  #include <regex>  std::string getValidatedInput() {  std::string input;  std::getline(std::cin, input);    // Validate input to contain only alphanumeric characters  if (!std::regex\_match(input, std::regex("^[a-zA-Z0-9]+$"))) {  std::cerr << "Invalid input! Only letters and numbers are allowed." << std::endl;  return "";  }    return input;  } |

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

| **Principles(s):**  **1. Validate Input Data** – Directly enforces the principle of validating user input before it’s processed, reducing risks like injection or buffer overflow. **7. Sanitize Data Sent to Other Systems** – Ensures sanitized input before passing it forward, preventing propagation of potentially malicious content. **8. Practice Defense in Depth** – Input validation is a first line of defense and helps enforce multiple layers of protection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-input | Flags unvalidated or unsanitized input. |
| SonarQube | 10.3 | cpp:S5131 | Detects improper handling of user input and recommends validation techniques. |
| Cppcheck | 2.13 | unvalidatedInput | Warns about unchecked input operations that could lead to unsafe use. |
| Visual Studio | 2022 | Code Analysis | Helps detect dangerous input patterns and use of deprecated input functions. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure File Handling | STD-009-C++ | Use Secure File Access Methods |

| **Noncompliant Code** |
| --- |
| Opening files without proper error handling can lead to security risks, such as unauthorized access or data corruption. |
| std::ifstream file("data.txt"); // No error handling |

| **Compliant Code** |
| --- |
| Checking if the file opened successfully prevents issues related to missing files, permission errors, or incorrect paths. |
| #include <iostream>  #include <fstream>  std::ifstream file("data.txt");  if (!file) {  std::cerr << "Error: Unable to open file!" << std::endl;  } else {  std::cout << "File opened successfully." << std::endl;  } |

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

| **Principles(s):**  **3. Architect and Design for Security Policies** – Incorporates secure design by ensuring files are accessed only when appropriate and available. **4. Keep It Simple** – Prevents ambiguous errors or undefined behavior by explicitly checking access success. **8. Practice Defense in Depth** – Adds a safeguard against one of the most common I/O attack vectors, forming part of a layered approach. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | fileAccessError | Warns when file streams are used without checking for open success. |
| Clang-Tidy | 15.0 | cppcoreguidelines-file-io | Ensures proper file I/O practices and error checking. |
| SonarQube | 10.3 | cpp:S3655 | Detects missing checks when opening files that may lead to runtime errors. |
| Visual Studio | 2022 | Code Analysis | Suggests proper use of file stream validation for reliability and security. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Using Deprecated Functions | STD-010-C++ | Use Modern C++ Functions Instead of Deprecated Ones |

| **Noncompliant Code** |
| --- |
| Using deprecated functions like gets() can lead to security vulnerabilities, such as buffer overflows. |
| char buffer[50];  gets(buffer); // Unsafe, deprecated function |

| **Compliant Code** |
| --- |
| Using std::getline() ensures safe input handling by preventing buffer overflows. |
| #include <iostream>  #include <string>  std::string buffer;  std::getline(std::cin, buffer); // Safe input handling |

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

| **Principles(s):**  **10. Adopt a Secure Coding Standard** – Modern functions are designed with safety in mind, and replacing deprecated functions aligns with industry best practices. **2. Heed Compiler Warnings** – Deprecated functions typically trigger compiler warnings, which should be resolved to avoid security flaws. **4. Keep It Simple** – Using standard and modern APIs simplifies maintenance and improves code clarity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | deprecatedFunction | Detects use of deprecated functions like gets() or strcpy(). |
| Clang-Tidy | 15.0 | modernize-deprecated-headers | Flags deprecated headers and functions to encourage use of modern equivalents. |
| SonarQube | 10.3 | cpp:S104 | Identifies use of unsafe and deprecated functions with secure alternatives recommended. |
| Visual Studio | 2022 | Code Analysis (C26440) | Warns against usage of insecure, deprecated, or obsolete functions. |

### 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.

Automation plays a key role in ensuring compliance with the security policy throughout Green Pace’s DevOps pipeline. The DevSecOps model promotes the integration of security measures at every stage of the development lifecycle, embedding enforcement and compliance directly into the automation process. Based on the provided diagram, security should not be an afterthought but a continuous, integrated process.

In the **Plan** phase, security policies—such as the C/C++ coding standards—should be reviewed and incorporated into development requirements and user stories. Automated tools like static analysis checkers (e.g., **Cppcheck**, **Clang-Tidy**) can be configured with custom rules to match the company’s coding standards. These rules should be stored in a version-controlled repository to ensure consistency across projects.

During **Develop** and **Build**, integration with code editors and CI tools ensures that noncompliant code is flagged early. Pre-commit hooks and build tools (like **CMake** or **Makefiles**) can run static analyzers automatically, rejecting builds that violate critical standards.

In the **Test** phase, dynamic analysis tools and unit test frameworks should be extended to include security-related test cases. Automation ensures these tests are always run, catching regressions or new vulnerabilities.

The **Release** and **Deploy** stages can include security scanning tools (e.g., **SonarQube**, **Fortify**, or **Coverity**) as part of the continuous delivery pipeline. Security gates can block deployments if the risk level or number of vulnerabilities exceeds acceptable thresholds.

During **Operate** and **Monitor**, runtime security tools and logging solutions help detect anomalies. Security configurations should be checked continuously using automated compliance-as-code tools like **OpenSCAP** or **Chef InSpec**.

By automating these processes, Green Pace ensures that the enforcement of coding standards and other security measures is consistent, repeatable, and efficient—minimizing manual error and improving overall security posture without slowing down development.

### 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 | Likely | Low | Medium | 3 |
| STD-002-CPP | High | Likely | Low | High | 4 |
| STD-003-CPP | High | Likely | Low | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 5 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 3 |
| STD-007-CPP | High | Unlikely | Medium | High | 3 |
| STD-008-CPP | High | Likely | Medium | High | 5 |
| STD-009-CPP | Medium | Likely | Low | Medium | 3 |
| STD-010-CPP | High | Likely | Low | 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 | **What it is**: Protecting stored data by converting it into a secure format.  **How**: Utilize strong encryption algorithms (e.g., AES-256) for databases, file systems, and backups.  **Why and When**: Ensures data confidentiality in case of physical theft or unauthorized access to storage media. Applicable to all stored sensitive information. |
| Encryption in flight | **What it is**: Securing data during transmission between systems.  **How**: Implement protocols like TLS/SSL for all data exchanges over networks.  **Why and When**: Prevents interception and tampering during data transfer. Essential for all network communications involving sensitive data. |
| Encryption in use | **What it is**: Protecting data while it's being processed in memory.  **How**: Employ techniques like homomorphic encryption or secure enclaves (e.g., Intel SGX).  **Why and When**: Safeguards data from memory scraping or unauthorized access during processing. Critical for handling highly sensitive computations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **What it is**: Verifying the identity of users or systems.  **How**: Implement multi-factor authentication (MFA), strong password policies, and biometric verification where applicable.  **Why and When**: Ensures only authorized individuals access the system. Applies to all user login processes. |
| Authorization | **What it is**: Granting users permission to access specific resources.  **How**: Use role-based access control (RBAC) to assign permissions based on user roles.  **Why and When**: Restricts access to sensitive data/functions to only those who need it. Applies during user role assignments and access requests. |
| Accounting | **What it is**: Tracking user activities within the system.  **How**: Maintain detailed logs of user logins, data access, changes to the database, addition of new users, and files accessed.  **Why and When**: Enables auditing, detects unauthorized activities, and supports compliance requirements. Continuous monitoring is essential. |

**\***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 | 03/25/2025 | Updated Coding Standards, revised version control process | Martin Garza |  |
| 3.0 | 04/12/2025 | Updated policy to align with Project One submission standards; improved automation section; revised version control process | Martin Garza |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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