**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 data entering the system is safe and meets expected formats. This prevents attackers from injecting malicious code or malformed data, which can exploit vulnerabilities like SQL injection or buffer overflows. Proper validation serves as a first line of defense against many types of attacks |
| 1. Heed Compiler Warnings | Compiler warnings often highlight potential issues in the code that could lead to vulnerabilities. Ignoring these warnings leaves the door open for attackers to exploit flaws. Developers should treat warnings as opportunities to improve code quality and security. |
| 1. Architect and Design for Security Policies | Security should be built into the architecture and design of systems from the beginning. By planning for security, developers can enforce consistent policies and minimize vulnerabilities. This proactive approach reduces the risks of retrofitting security measures later. |
| 1. Keep It Simple | Simple designs are easier to understand, test, and secure compared to overly complex systems. Complexity often introduces unintended security gaps and makes maintenance difficult. A streamlined system reduces the chances of human error and hidden vulnerabilities. |
| 1. Default Deny | Access to systems and resources should be denied by default, only granting permissions explicitly when necessary. This principle ensures that unauthorized actions are blocked, reducing the attack surface. It enforces a more cautious and secure posture. |
| 1. Adhere to the Principle of Least Privilege | This principle restricts users and processes to only the permissions they need to perform their tasks. Limiting privileges reduces the impact of errors or compromised accounts. It is an essential safeguard against privilege escalation attacks. |
| 1. Sanitize Data Sent to Other Systems | Data shared with external systems should be sanitized to remove potentially harmful content. This protects against injection attacks and ensures the receiving system functions as intended. Proper sanitation also maintains data integrity and security. |
| 1. Practice Defense in Depth | Defense in depth employs multiple layers of security to protect a system. If one layer is compromised, others remain in place to provide protection. This approach minimizes the chances of a full-scale breach and increases resilience. |
| 1. Use Effective Quality Assurance Techniques | Comprehensive testing and quality assurance practices identify and fix vulnerabilities before deployment. Techniques like static analysis, code reviews, and automated testing ensure that the code meets security standards. Regular QA processes improve the reliability and security of software. |
| 1. Adopt a Secure Coding Standard | Secure coding standards provide guidelines for writing secure and consistent code. Following these standards reduces the risk of introducing vulnerabilities during development. They are essential for maintaining security best practices across teams and projects. |

### 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-101-CPP | **Use Appropriate Data Types to Prevent Overflow and Data Loss**  Using incorrect or inadequate data types can result in overflow, underflow, or data truncation, leading to incorrect program behavior or security vulnerabilities. Selecting appropriate data types ensures that data is stored and manipulated safely, preserving its integrity and preventing unexpected behavior. |

| **Noncompliant Code** |
| --- |
| Assigning a large integer value to a variable of insufficient size causes integer overflow. |
| #include <iostream>  int main() {  unsigned short smallNumber = 70000; // exceeds max value for unsigned short  std::cout << "smallNumber = " << smallNumber << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using a larger data type accommodates the large integer value without causing overflow. |
| #include <iostream>  int main() {  unsigned long smallNumber = 70000; // unsigned long can hold larger values  std::cout << "smallNumber = " << smallNumber << std::endl;  return 0;  } |

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

| **Principles(s):** 1,9,10  Validate Input Data: Ensures values fit correctly with chosen data types.  Use Effective Quality Assurance Techniques: QA processes can detect type-related errors early.  Adopt a Secure Coding Standard: Correct data types are a fundamental part of secure coding guidelines. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.16.0 | Integer overflow checks | Identifies potential integer overflows |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-220-CPP | **Ensure Data Values Are Within Expected Ranges**  Processing data values outside their expected ranges can lead to incorrect program behavior or security vulnerabilities such as buffer overflows or logic errors. Validating data ranges before use ensures program correctness and security. |

| **Noncompliant Code** |
| --- |
| The code does not check if the input number is within the expected range, potentially causing unexpected behavior. |
| #include <iostream>  void processData(int index) {  int data[10];  data[index] = 0; // no check on index  }  int main() {  int userIndex;  std::cin >> userIndex;  processData(userIndex);  return 0;  } |

| **Compliant Code** |
| --- |
| The code validates the input to ensure it falls within the valid range before using it. |
| #include <iostream>  void processData(int index) {  int data[10];  if (index >= 0 && index < 10) {  data[index] = 0;  } else {  // handle error  std::cerr << "Index out of bounds" << std::endl;  }  }  int main() {  int userIndex;  std::cin >> userIndex;  processData(userIndex);  return 0;  } |

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

| **Principles(s):** 1,9,10  Validate Input Data: Ensuring values fall within expected ranges is core to input validation.  Use Effective Quality Assurance Techniques: Range checks are easily tested and caught by QA.  Adopt a Secure Coding Standard: Range validation is a fundamental coding standard practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Array index bounds checking | Flags out-of-bound array access |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-310-CPP | **Avoid Buffer Overflows When Handling Strings**  Improper handling of strings can lead to buffer overflows, which are a common source of security vulnerabilities. Ensuring that string operations do not exceed buffer sizes prevents overwriting memory and potential code execution by attackers. |

| **Noncompliant Code** |
| --- |
| The code uses strcpy without checking the destination buffer size, which may cause a buffer overflow. |
| #include <cstring>  void copyString(char\* dest, const char\* src) {  strcpy(dest, src); // no length check  }  int main() {  char buffer[10];  const char\* longString = "This string is too long";  copyString(buffer, longString);  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses strncpy and ensures the destination buffer is not overrun. |
| #include <cstring>  void copyString(char\* dest, const char\* src, size\_t destSize) {  strncpy(dest, src, destSize - 1);  dest[destSize - 1] = '\0'; // ensure null termination  }  int main() {  char buffer[10];  const char\* longString = "This string is too long";  copyString(buffer, longString, sizeof(buffer));  return 0;  } |

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

| **Principles(s):** 1,7,9,10  Validate Input Data: Ensuring string length matches buffer size.  Sanitize Data Sent to Other Systems: Prevents malicious input from overflowing buffers.  Use Effective Quality Assurance Techniques: Automated tests can catch unsafe string operations.  Adopt a Secure Coding Standard: Secure handling of strings is a standard coding practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Buffer Overflow | Detects unsafe string copy and overflows |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-440-CPP | **Prevent SQL Injection by Using Parameterized Queries**  Constructing SQL queries by concatenating user input can lead to SQL injection vulnerabilities. Using parameterized queries ensures that user input is treated as data rather than executable code, preventing attackers from altering the intended SQL commands. |

| **Noncompliant Code** |
| --- |
| The code constructs an SQL query by concatenating user input directly, making it vulnerable to SQL injection. |
| #include <string>  #include <iostream>  void getUserInfo(const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'";  }  int main() {  std::string name;  std::cin >> name;  getUserInfo(name);  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses parameterized queries to safely include user input in the SQL command. |
| #include <string>  #include <iostream>  #include <sqlite3.h>  void getUserInfo(const std::string& userInput) {  sqlite3\* db;  sqlite3\_open("users.db", &db);  const char\* sql = "SELECT \* FROM users WHERE name = ?";  sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_TRANSIENT);  sqlite3\_finalize(stmt);  sqlite3\_close(db);  }  int main() {  std::string name;  std::cin >> name;  getUserInfo(name);  return 0;  } |

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

| **Principles(s):** 1,7,8, 10  Validate Input Data: Ensures user input does not become executable code.  Sanitize Data Sent to Other Systems: Parameterization ensures sanitized input for databases.  Practice Defense in Depth: Multiple layers (validation, parameterization) protect the system.  Adopt a Secure Coding Standard: Using parameterized queries is a recognized secure standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Suspicious string concatenation | Flags direct SQL query concatenation |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | |  | | --- | | STD-515-CPP |  |  | | --- | |  | | **Avoid Memory Leaks and Dangling Pointers**  Improper memory management can lead to memory leaks or use of invalid memory, causing program instability or vulnerabilities. Proper allocation, deallocation, and use of smart pointers can prevent such issues. |

| **Noncompliant Code** |
| --- |
| The code allocates memory but does not deallocate it, leading to a memory leak. |
| #include <iostream>  void allocateMemory() {  int\* data = new int[100]; // missing delete[]  }  int main() {  allocateMemory();  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses smart pointers to manage memory automatically, ensuring proper deallocation. |
| #include <iostream>  #include <memory>  void allocateMemory() {  std::unique\_ptr<int[]> data(new int[100]);  // memory is automatically deallocated when data goes out of scope  }  int main() {  allocateMemory();  return 0;  } |

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

| **Principles(s):** 2,4,8,9, 10  Heed Compiler Warnings: Compilers often warn about memory issues.  Keep It Simple: Smart pointers simplify memory management.  Practice Defense in Depth: Proper memory hygiene reduces attack vectors.  Use Effective Quality Assurance Techniques: Tools and tests catch leaks.  Adopt a Secure Coding Standard: Following guidelines for memory management is standard practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Memory leak | Detects allocated memory not freed. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-660-CPP | **Use Assertions to Document and Enforce Assumptions**  Assertions help in catching programming errors by checking for conditions that should never occur during execution. They act as internal self-checks, ensuring that code behaves as expected. |

| **Noncompliant Code** |
| --- |
| The code assumes that a pointer is not null without checking, which can lead to a null pointer dereference. |
| #include <iostream>  void printValue(int\* ptr) {  std::cout << \*ptr << std::endl; // no null check  }  int main() {  int\* value = nullptr;  printValue(value);  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses an assertion to ensure the pointer is not null before dereferencing. |
| #include <iostream>  #include <cassert>  void printValue(int\* ptr) {  assert(ptr != nullptr);  std::cout << \*ptr << std::endl;  }  int main() {  int\* value = nullptr;  printValue(value); // will trigger assertion failure  return 0;  } |

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

| **Principles(s):** 1,4,9, 10  Validate Input Data: Ensuring assumptions about inputs and pointers are correct.  Keep It Simple: Assertions are a simple mechanism to prevent obscure bugs.  Use Effective Quality Assurance Techniques: Assertions help QA identify logic flaws.  Adopt a Secure Coding Standard: Assertions are recommended in secure coding guidelines. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Unchecked pointer usage | Flags use of pointers without checks |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | |  | | --- | | STD-715-CPP |  |  | | --- | |  | | **Handle Exceptions Appropriately to Maintain Program Stability**  Exceptions provide a mechanism for handling errors. Properly catching and handling exceptions prevents the program from crashing and allows for graceful recovery or termination. |

| **Noncompliant Code** |
| --- |
| The code does not handle exceptions that may be thrown, leading to possible abrupt program termination. |
| #include <iostream>  #include <vector>  void processData() {  std::vector<int> data;  data.at(5) = 10; // may throw out\_of\_range error  }  int main() {  processData();  return 0;  } |

| **Compliant Code** |
| --- |
| The code catches exceptions and handles them appropriately. |
| #include <iostream>  #include <vector>  void processData() {  try {  std::vector<int> data;  data.at(5) = 10;  } catch (const std::out\_of\_range& e) {  std::cerr << "Out of range error: " << e.what() << std::endl;  // handle error  }  }  int main() {  processData();  return 0;  } |

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

| **Principles(s):** 3,9, 10  Architect and Design for Security Policies: Robust error handling is part of secure design.  Use Effective Quality Assurance Techniques: QA tests exception handling paths.  Adopt a Secure Coding Standard: Handling exceptions is a fundamental secure coding practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Exception safety checks | Identifies unhandled exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Const Correctness | STD-801-CPP | **Use const Correctly to Protect Immutable Data**  Using const correctly ensures that variables intended to be immutable cannot be modified after initialization. This prevents unintended side effects and enhances code safety by making the code's intentions clear, reducing the likelihood of bugs and security vulnerabilities due to accidental modification of data. |

| **Noncompliant Code** |
| --- |
| The code modifies a variable that should be constant, potentially leading to unexpected behavior. |
| #include <iostream>  void printMessage(char\* message) {  message[0] = 'H'; // modifying the message  std::cout << message << std::endl;  }  int main() {  const char\* msg = "hello";  printMessage(const\_cast<char\*>(msg));  return 0;  } |

| **Compliant Code** |
| --- |
| The code correctly uses const to prevent modification of immutable data. |
| #include <iostream>  void printMessage(const char\* message) {  // message[0] = 'H'; // not allowed, compiler will prevent this  std::cout << message << std::endl;  }  int main() {  const char\* msg = "hello";  printMessage(msg);  return 0;  } |

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

| **Principles(s):** 4, 9, 10  Keep It Simple: Const usage clarifies intent and simplifies debugging.  Use Effective Quality Assurance Techniques: QA can verify compliance easily.  Adopt a Secure Coding Standard: Proper const usage is a recognized best practice. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Priority | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Const correctness | Flags variables that could be const |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Proper Initialization | STD-009-CPP | **Initialize Variables Before Use**  Using uninitialized variables can lead to undefined behavior, crashes, or security vulnerabilities due to unpredictable values. Initializing variables before use ensures that they hold known, valid values, enhancing program stability and security. |

| **Noncompliant Code** |
| --- |
| The code uses an uninitialized variable, leading to undefined behavior. |
| #include <iostream>  int main() {  int count;  if (count > 0) { // count is uninitialized  std::cout << "Count is positive." << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| The code initializes the variable before use, ensuring predictable behavior. |
| #include <iostream>  int main() {  int count = 0;  if (count > 0) {  std::cout << "Count is positive." << std::endl;  }  return 0;  } |

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

| **Principles(s):** 1,2,9, 10  Validate Input Data: Initialization ensures data starts in a known state.  Heed Compiler Warnings: Compilers often warn about uninitialized variables.  Use Effective Quality Assurance Techniques: Easy to test for uninitialized variables.  Adopt a Secure Coding Standard: Initializing variables is a basic secure practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Uninitialized variable | Detects variables used without init |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Safe Functions | STD-190-CPP | **Avoid Using Deprecated or Unsafe Functions**  Deprecated or unsafe functions may have known security vulnerabilities or may be removed in future versions of the language or libraries. Avoiding these functions helps prevent security issues and ensures code remains maintainable and compatible. |

| **Noncompliant Code** |
| --- |
| The code uses gets(), which is deprecated and unsafe due to its inability to prevent buffer overflows. |
| #include <cstdio>  int main() {  char buffer[10];  gets(buffer); // unsafe function  printf("You entered: %s\n", buffer);  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses fgets(), which allows specifying the buffer size, preventing overflows |
| #include <cstdio>  int main() {  char buffer[10];  if (fgets(buffer, sizeof(buffer), stdin) != NULL) {  printf("You entered: %s\n", buffer);  }  return 0;  } |

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

| **Principles(s):** 2,9, 10  Heed Compiler Warnings: Deprecated functions often generate warnings.  Use Effective Quality Assurance Techniques: QA easily identifies unsafe calls.  Adopt a Secure Coding Standard: Following standards means avoiding unsafe functions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.16.0 | Deprecated function use | Identifieds calls to functions like gets() |

### 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.

To align Green Pace's DevOps process with the policy, automation should be integrated across the DevSecOps pipeline for seamless enforcement and compliance. In pre-production, automated tools can assess threats, enforce design standards like OWASP, validate secure builds from trusted sources, and perform thorough security testing through vulnerability scans. During production, automation ensures consistent configurations, proactive monitoring with SIEM tools, and rapid incident responses like blocking attacks or rolling back changes. Post-incident, automated baseline assessments confirm systems are secure and stable. This approach embeds compliance into every phase, enhancing security while maintaining operational efficiency.

### 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-101-CPP | High | Likely | Low | High | 4 |
| STD-220-CPP | High | Likely | Low | High | 4 |
| STD-310-CPP | High | Likely | Medium | High | 5 |
| STD-440-CPP | High | Likely | Medium | High | 5 |
| STD-515-CPP | Medium | Likely | Low | High | 3 |
| STD-660-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-715-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-801-CPP | Low | Likely | Low | Low | 1 |
| STD-009-CPP | Medium | Likely | Low | High | 3 |
| STD-190-CPP | Medium | Likely | Low | High | 3 |

### 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 persistent media from unauthorized access if the storage medium is compromised. By encrypting files, databases, and backups, unauthorized parties who obtain physical access to hardware or snapshots will not be able to read the sensitive data. This policy applies to all sensitive or regulated data stored by Green Pace. It must be enforced whenever data is saved to long-term storage. |
| Encryption in flight | Encryption in flight ensures that data is protected while moving between systems, services, or networks. Transport Layer Security (TLS) or similar encryption methods are used so that eavesdroppers cannot access sensitive information as it travels over insecure networks such as the internet. This policy applies to all data leaving secure boundaries and crossing networks. It is mandatory for all external API calls, data replication, and user communications. |
| Encryption in use | Encryption in use safeguards data while it is being actively processed in memory. It is a more complex and resource-intensive process, but for highly sensitive information, it ensures that data never exists in plaintext, not even temporarily in system memory. This policy applies to operations handling the most sensitive data types where exposure in RAM could pose unacceptable risks. Examples include cryptographic key management or extremely sensitive personally identifiable information (PII). |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users or systems before granting access to resources. This involves user credentials or integrated identity management solutions. The policy mandates strong, multifactor authentication for all critical systems. It applies whenever a user attempts to log in to a system, access protected services, or perform actions requiring identity verification. Proper authentication ensures only legitimate users gain entry into Green Pace systems. |
| Authorization | Authorization ensures that authenticated users have only the privileges necessary to perform their tasks. The policy mandates roles, permissions, and access controls that map directly to job functions. Authorization checks are performed upon each request to sensitive operations, API endpoints, or protected resources. This guarantees that even if a user is authenticated, they cannot exceed their intended level of access. |
| Accounting | Accounting, often referred to as auditing, tracks user and system activities, creating logs for all operations performed. This policy requires maintaining detailed records of user logins, file accesses, database changes, and the addition or removal of users. Audit logs must be tamper-evident and retained for a defined period. These records are essential for compliance audits, forensic investigations, and internal reviews to ensure that established security policies are being followed. |

**\***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 |  |
| 1.1 | 11/17/2024 | Milestone One | Darius Stewart |  |
| 1.2 | 12/8/2024 | Final Security Policy | Darius Stewart |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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