**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 | Always validate user-provided data to ensure it meets expected formats and ranges. This prevents injection attacks like SQL injection, buffer overflows, and other vulnerabilities caused by malformed or malicious input. |
| 1. Heed Compiler Warnings | Pay attention to compiler warnings, as they often indicate potential issues such as uninitialized variables, type mismatches, or memory-related bugs. Configure compilers to treat warnings as errors to ensure code quality and prevent latent vulnerabilities. |
| 1. Architect and Design for Security Policies | Incorporate security considerations at the design phase of development. Build systems with well-defined policies for authentication, authorization, data integrity, and resilience against known attack vectors. |
| 1. Keep It Simple | Avoid overly complex designs, as they are harder to understand, debug, and secure. Simplicity reduces the likelihood of introducing vulnerabilities and makes maintenance easier over time. |
| 1. Default Deny | Implement default-deny policies for access controls. Only explicitly permitted actions or users should gain access, minimizing the risk of unauthorized operations and data breaches. |
| 1. Adhere to the Principle of Least Privilege | Assign the minimum level of access or permissions necessary for a user or process to perform its function. This limits potential damage in case of a breach or misuse. |
| 1. Sanitize Data Sent to Other Systems | Ensure that data sent to external systems is properly sanitized and conforms to expected formats. This prevents injection attacks and ensures compatibility with the target system. |
| 1. Practice Defense in Depth | Employ multiple layers of security to protect systems and data. If one layer is breached, additional layers (e.g., firewalls, encryption, authentication) act as safeguards, reducing the overall risk. |
| 1. Use Effective Quality Assurance Techniques | Implement rigorous quality assurance practices such as code reviews, static analysis, and automated testing to identify and address vulnerabilities early in the development lifecycle. |
| 1. Adopt a Secure Coding Standard | Follow a secure coding standard, such as SEI CERT C++, to ensure consistent implementation of best practices. Secure coding standards provide guidelines for preventing common vulnerabilities and maintaining system integrity. |

### 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 | **Validate Input Length**: input length is validated prevents buffer overflows, which can lead to security vulnerabilities, such as overwriting adjacent memory or executing arbitrary code. Proper input validation is a foundational practice in secure coding. |

| **Noncompliant Code** |
| --- |
| Does not check the input length, leading to potential overflow. |
| char buffer[10];  std::cin >> buffer; // No size check |

| **Compliant Code** |
| --- |
| Ensures input does not exceed buffer size. |
| char buffer[10];  std::cin.get(buffer, sizeof(buffer)); // Size is checked |

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

| **Principles(s):** **Validate Input Data:** ensures data remains within expected constraints preventing potential buffer overflows and malicious exploits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | Bufferoverflow | Detects potential buffer overflows due to unchecked input. |
| Clang-Tidy | 15.0.6 | clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling | Flags unsafe input handling functions like cin >> buffer. |
| SonarQube | 9.9 LTS | CEW-120 | Detects C++ buffer overflow vulnerabilities |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Validating Data Values:** ensures inputs and variables fall within acceptable ranges, preventing errors like integer overflows, underflows, or unexpected behavior in computations. This practice improves code reliability and security by detecting and mitigating invalid data usage. |

| **Noncompliant Code** |
| --- |
| Does not check if a value is within an expected range, leading to potential overflow or incorrect results. |
| int divisor = user\_input; // Assume user\_input is 0  int result = 100 / divisor; // Risk of division by zero |

| **Compliant Code** |
| --- |
| Ensures the value is within an expected range before performing operations. |
| int divisor = user\_input;  if (divisor != 0) {  int result = 100 / divisor; // Safe operation  std::cout << "Result: " << result << std::endl;  } else {  std::cerr << "Error: Division by zero is not allowed." << std::endl;  } |

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

| **Principles(s):** **Validating Data Values:** Checking data before using it prevents unexpected errors such as division by zero, overflow, or system crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | unvalidatedInput | Detects unvalidated input leading to potential division by zero |
| Clang-Tidy | 15.0.6 | clang-analyzer-core.DivideZero | Flags cases where division by zero is possible. |
| SonarQube | 9.9 LTS | CWE-369 | Identifies division-by-zero vulnerabilities. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | **Ensure String Correctness:** String correctness ensures proper handling, validation, and manipulation of strings to prevent vulnerabilities like buffer overflows, null-termination issues, and injection attacks. |

| **Noncompliant Code** |
| --- |
| Fails to validate string length or ensure null-termination, leading to potential buffer overflows or undefined behavior. |
| char name[10];  std::cin >> name; // Unsafe: Does not validate input size |

| **Compliant Code** |
| --- |
| Ensures the string is safely handled by validating input length and ensuring null-termination. |
| char name[10];  std::cin.get(name, sizeof(name)); //Limits input to fit buffer  name[sizeof(name) - 1] = '\0'; // Ensures null-termination  std::cout << "Hello, " << name << std::endl; |

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

| **Principles(s):** **Ensure String Correctness:** Improper string handling can **cause buffer overflows, data corruption, or injection attacks**. Enforcing **input length validation** ensures data integrity and prevents exploits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | Bufferoverflow | Detects improper buffer handling that could cause overflows. |
| Clang-Tidy | 15.0.6 | |  | | --- | |  |  |  | | --- | | clang-analyzer-security.insecureAPI.strcpy | | Flags unsafe string functions like gets() and strcpy(). |
| SonarQube | 9.9 LTS | CWE-120 | Identifies buffer overflows and improper input handling. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **Prevent SQL Injection:** occur when user inputs are directly embedded in SQL queries without validation or sanitization. |

| **Noncompliant Code** |
| --- |
| Directly concatenates user input into the SQL query, making it vulnerable to SQL injection attacks. |
| std::string username = user\_input; // Assume user\_input = "admin'; // DROP TABLE USERS;--"  std::string query = "SELECT \* FROM USERS WHERE USERNAME = '" + username + "';";  database.execute(query); // Vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| Uses parameterized queries to safely handle user input, ensuring input is properly sanitized. |
| std::string username = user\_input;  std::string query = "SELECT \* FROM USERS WHERE USERNAME = ?";  PreparedStatement stmt = database.prepare(query);  stmt.bind(1, username); // Bind user input as a parameter  stmt.execute(); // Safely executes the query |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems:** Directly inserting user input into SQL statements can lead to injection attacks. Using parameterized queries ensures safe, sanitized interactions with the database. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.12.1 | sqlInjection | Detects direct user input concatenation in SQL queries. |
| SonarQube | 9.9 LTS | S3649 | Flags SQL queries that are constructed using unsanitized user input. |
| Bandit (Python) | 1.7.5 | B608 | Identifies SQL injection vulnerabilities in Python applications. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **Ensure Memory Protection:** memory is allocated, accessed, and deallocated safely, preventing issues like buffer overflows, dangling pointers, double deletions, or memory leaks. This standard helps avoid undefined behavior, protects sensitive data, and maintains application stability and security. |

| **Noncompliant Code** |
| --- |
| Improper memory management resulting in a dangling pointer. |
| int\* ptr = new int(42);  delete ptr; // Memory is freed  \*ptr = 10; // Undefined behavior: accessing a dangling pointer |

| **Compliant Code** |
| --- |
| Uses smart pointers to ensure memory is automatically managed and avoids dangling pointers. |
| #include <memory>  std::unique\_ptr<int> ptr = std::make\_unique<int>(42); // Memory is //automatically managed  \*ptr = 10; // Safe access  // Memory is freed when ptr goes out of scope |

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

| **Principles(s):** **Adhere to the Principle of Least Privilege:** Proper memory management ensures memory safety, limiting unintended access or modification. Prevents exploits like heap overflow attacks, which can lead to arbitrary code execution.  **Use Effective Quality Assurance Techniques:** Modern C++ features like smart pointers reduce human error and enhance security. Automated tools can detect raw pointer misuse and recommend safe alternatives. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | memleak, uninitvar | Detects memory leaks and uninitialized pointer access. |
| Clang-Tidy | 17.0.1 | modernize-use-std-smart-ptr | Flags raw pointers that should be replaced with smart pointers. |
| Valgrind | 3.19.0 | Memcheck | Identifies memory leaks, uninitialized reads, and invalid accesses. |
| AddressSanitizer (ASan) | Built-In | -fsanitize=address | Detects out-of-bounds memory accesses and use-after-free errors. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Use Assertions to Verify Assumptions:** Assertions validate assumptions made by the program during runtime and help catch logical errors in development and testing phases. They should only be used for debugging purposes, as they can provide early detection of violations in code behavior or invalid inputs. Assertions should not replace error handling in production environments. |

| **Noncompliant Code** |
| --- |
| Fails to validate assumptions, leading to potential undefined behavior. |
| int divisor = user\_input; // Assume user\_input is 0  int result = 100 / divisor; // Potential division by zero |

| **Compliant Code** |
| --- |
| Uses assertions to verify critical assumptions during development and testing. |
| #include <cassert>  int divisor = user\_input;  assert(divisor != 0 && "Divisor must not be zero"); // Ensures // divisor is valid  int result = 100 / divisor; // Safe operation |

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

| **Principles(s):** **Validate Input Data:** Assertions ensure assumptions hold true, preventing invalid input propagation.  **Heed Compiler Warnings:** Many compilers can detect potential assertion failures, guiding developers toward correct usage.  **Use Effective Quality Assurance Techniques:** Assertions catch errors early, reducing debugging time and improving code reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | assertWithSideEffect | Flags assertions that contain side effects. |
| Clang-Tidy | 17.0.1 | |  | | --- | |  |  |  | | --- | | misc-static-assert | | Detects assertion misuse and suggests static\_assert where applicable. |
| PVS-Studio | 7.27.0 | V501, V512 | Identifies missing assertions and checks for incorrect error handling. |
| AddressSanitizer (ASan) | Built-in | |  | | --- | |  |  |  | | --- | | -fsanitize=undefined | | Detects runtime errors like division by zero or buffer overflows. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | **Use Exceptions for Error Handling:** provide a structured mechanism for handling errors and exceptional conditions in a program. |

| **Noncompliant Code** |
| --- |
| Uses return codes for error handling, which can result in ignored errors or inconsistent handling. |
| int divide(int numerator, int denominator) {  if (denominator == 0) {  return -1; // Error code  }  return numerator / denominator;  }  int result = divide(10, 0);  if (result == -1) {  std::cerr << "Error: Division by zero" << std::endl;  } |

| **Compliant Code** |
| --- |
| Uses exceptions to handle errors, ensuring the error cannot be ignored. |
| int divide(int numerator, int denominator) {  if (denominator == 0) {  throw std::invalid\_argument("Division by zero is not allowed");  }  return numerator / denominator;  }  try {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  } |

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

| **Principles(s):** **Adhere to the Principle of Least Privilege:** Exceptions reduce unnecessary permissions for error-handling functions, improving security.  **Practice Defense in Depth**: Ensures multiple layers of error handling prevent application crashes.  **Use Effective Quality Assurance Techniques**: Exception handling improves code reliability, preventing silent failures. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | |  | | --- | |  |  |  | | --- | | missingThrowSpecifier | | Detects functions that should throw exceptions but don't. |
| Clang-Tidy | 17.0.1 | bugprone-exception-escape | |  | | --- | |  |  |  | | --- | | Identifies exceptions escaping destructors. | |
| PVS-Studio | 7.27.0 | V596 | |  | | --- | |  |  |  | | --- | | Warns about improper exception handling. | |
| SonarQube | 9.9.0 | Cpp:S134 | |  | | --- | |  |  |  | | --- | | Detects uncaught exceptions and improper error handling. | |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Dangerous Functions | STD-008-CPP | **Avoid Dangerous Functions:** Certain C/C++ functions, such as gets and strcpy, are inherently unsafe as they do not perform bounds checking, leading to potential buffer overflows and other vulnerabilities. Avoiding these functions and using safer alternatives helps ensure memory safety and prevents security issues. |

| **Noncompliant Code** |
| --- |
| Uses strcpy, which can lead to a buffer overflow if the source string is larger than the destination buffer. |
| #include <cstring>  char dest[10];  strcpy(dest, "This is too long!"); // Unsafe: No bounds checking |

| **Compliant Code** |
| --- |
| Uses strncpy, a safer alternative that limits the number of characters copied. |
| #include <cstring>  char dest[10];  strncpy(dest, "This is too long!", sizeof(dest) - 1); // Safe: Copies up to buffer size minus 1  dest[sizeof(dest) - 1] = '\0'; // Ensures null-termination |

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

| **Principles(s):** **Validate Input Data**: Using unsafe functions without validation allows buffer overflows.  **Principle: Adhere to the Principle of Least Privilege**: functions can give attackers unintended access to memory.  **Principle: Use Effective Quality Assurance Techniques**: Automated tools can detect unsafe functions and suggest safer alternatives. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | strcpy detection | Detects unsafe functions like strcpy() and gets(). |
| Clang-Tidy | 17.0.1 | cert-err34-c | |  | | --- | |  |  |  | | --- | | Identifies unsafe string handling. | |
| PVS-Studio | 7.27.0 | V512 | |  | | --- | |  |  |  | | --- | | Warns about potential buffer overflows. | |
| SonarQube | 9.9.0 | Cpp:S3519 | Detects unsafe memory manipulation. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Random Number Generation | STD-009-CPP | **Random Number Generation:** common requirement in many applications, including cryptographic systems, simulations, and gaming. Using insecure or predictable random number generators like rand() can result in weak security and predictable behavior. Adopting secure alternatives ensures stronger randomness and enhances application security. |

| **Noncompliant Code** |
| --- |
| Uses rand(), which is not cryptographically secure and can produce predictable results. |
| #include <cstdlib>  #include <ctime>  std::srand(std::time(nullptr)); // Seed with current time  int random\_number = std::rand(); // Predictable random number |

| **Compliant Code** |
| --- |
| Uses a secure random number generator such as std::random\_device and modern C++ random utilities. |
| #include <random>  std::random\_device rd; // Secure random number generator  std::mt19937 gen(rd()); // Mersenne Twister generator  std::uniform\_int\_distribution<> dist(1, 100); // Generate numbers in // range [1, 100]  int random\_number = dist(gen); // Secure random number |

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

| **Principles(s):** **Validate Input Data:** Using weak random generators leads to predictable results, making encryption and authentication vulnerable.  **Principle: Use Effective Quality Assurance Techniques**: Automated tools can detect insecure random number usage.  **Principle: Adopt a Secure Coding Standard**: Secure coding standards require using cryptographically strong randomness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | rand() detection | Detects usage of rand() and suggests alternatives. |
| Clang-Tidy | 17.0.1 | cert-msc50-cpp | Warns about insecure random number generation. |
| PVS-Studio | 7.27.0 | V1021 | Identifies predictable random number usage. |
| SonarQube | 9.9.0 | Cpp:S6418 | Detects weak reandomness in security-sensitive contexts |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Hardcoded Sensitive Data | STD-010-CPP | **Avoid Hardcoding Sensitive Data:** sensitive data, such as passwords, API keys, or encryption keys, into the source code makes the application vulnerable to attacks. If the source code is exposed, attackers can easily extract the sensitive information. Instead, use secure configuration files, environment variables, or secrets management tools to protect sensitive data. |

| **Noncompliant Code** |
| --- |
| Hardcodes sensitive data into the source code, making it vulnerable to exposure. |
| const std::string API\_KEY = "hardcodedapikey123"; // Hardcoded // sensitive data  void connectToService() {  std::cout << "Connecting with API key: " << API\_KEY << std::endl;  } |

| **Compliant Code** |
| --- |
| Uses environment variables or a secure configuration file to manage sensitive data. |
| void connectToService() {  const char\* api\_key = std::getenv("API\_KEY"); // Fetch from // environment variable  if (api\_key) {  std::cout << "Connecting with API key: " << api\_key << std::endl;  } else {  std::cerr << "Error: API key not set in environment variables!" << std::endl;  }  } |

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

| **Principles(s):** **Default Deny**: Hardcoded credentials grant unintended access if exposed. Using environment variables or secure storage limits exposure.  **Principle: Architect and Design for Security Policies**: Applications should be designed to separate configuration and logic.  **Principle: Sanitize Data Sent to Other Systems**: Secure handling prevents accidental exposure of sensitive credentials. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | hardcoded-credentials | Detects hardcoded API keys, passwords, and secrets. |
| Clang-Tidy | 17.0.1 | |  | | --- | |  |  |  | | --- | | cert-env01-cpp | | Warns about storing secrets in source code. |
| PVS-Studio | 7.27.0 | V1025 | |  | | --- | |  |  |  | | --- | | Identifies sensitive data hardcoded in applications. | |  | |  | |
| SonarQube | 9.9.0 | cpp:S5693 | |  | | --- | |  |  |  | | --- | | Flags hardcoded passwords in C++ projects. | |

### 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 make sure Green Pace follows the secure coding standards in this policy, automation will be built into multiple stages of the DevSecOps pipeline. The goal is to catch security issues early, reduce manual work, and make sure compliance is enforced throughout development. In the pre-production phase, tools like SonarQube, Cppcheck, and Clang-Tidy will handle static code analysis, flagging vulnerabilities before the code is even compiled. Software Composition Analysis (SCA) tools such as Snyk or OWASP Dependency-Check will scan open-source dependencies for known security flaws. On top of that, Dynamic Application Security Testing (DAST) tools like Burp Suite and OWASP ZAP will simulate attacks to catch runtime vulnerabilities. Automation will also be enforced through pre-commit hooks and IDE security plugins, ensuring developers fix security issues before pushing code.

Once the application moves to production, automation will shift toward monitoring, detection, and response. SIEM tools like Splunk and ELK Stack will track logs for unusual activity, while Intrusion Detection Systems (IDS) like Snort or Suricata will monitor for potential threats in real-time. If an attack is detected, automated response systems will help block threats, roll back compromised services, and restore a secure state. Regular security audits using tools like OpenSCAP will make sure Green Pace stays compliant with security policies. By integrating automation at each stage, Green Pace can stay secure while keeping up with fast development cycles, ensuring security is handled proactively rather than reactively.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Possible | Medium | High | 3 |
| STD-003-CPP | Medium | Possible | Low | Medium | 2 |
| STD-004-CPP | High | Likely | High | Critical | 5 |
| STD-005-CPP | High | Unlikely | High | High | 3 |
| STD-006-CPP | Medium | Possible | Low | Medium | 2 |
| STD-007-CPP | High | Likely | High | Critical | 5 |
| STD-008-CPP | High | Possible | Medium | High | 4 |
| STD-009-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-010-CPP | High | Likely | High | Critical | 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 | Encryption at rest refers to the protection of stored data from unauthorized access. This ensures that even if an attacker gains access to the physical storage medium, the data remains unreadable without the appropriate decryption keys. It is applied to databases, file storage, and backups to protect sensitive information. Green Pace enforces encryption at rest using AES-256 encryption for databases and sensitive files. Access to decryption keys is managed through a secure key management system (KMS), ensuring compliance with industry standards. |
| Encryption in flight | Encryption in flight protects data while it is being transmitted over a network, preventing interception and unauthorized access. This applies to data traveling between clients and servers, internal system communications, and APIs. Green Pace requires TLS 1.2 or higher for all communications, enforcing HTTPS, secure WebSocket connections, and VPN encryption where applicable. This policy ensures that data remains confidential and intact while in transit, safeguarding against man-in-the-middle attacks. |
| Encryption in use | Encryption in use secures data while it is actively being processed in memory. This is critical for protecting sensitive information from exposure during execution. Green Pace utilizes secure enclaves and memory encryption technologies such as Intel SGX or AMD SEV to prevent unauthorized access to sensitive computations. Applications handling personally identifiable information (PII) or cryptographic keys must use these methods to maintain confidentiality during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that only authorized users or systems can access Green Pace’s infrastructure and applications. Multi-factor authentication (MFA) is required for all system access, and strong password policies are enforced to mitigate brute-force attacks. Green Pace implements centralized authentication using identity providers (IdPs) such as OAuth 2.0, OpenID Connect, or SAML for single sign-on (SSO), ensuring a secure and unified authentication process. |
| Authorization | Authorization controls what authenticated users or systems can do within the environment. Green Pace follows the principle of least privilege (PoLP), ensuring users and applications have only the minimum permissions necessary to perform their tasks. Role-based access control (RBAC) and attribute-based access control (ABAC) are implemented across systems to enforce fine-grained access control policies. Periodic access reviews are conducted to prevent privilege creep and unauthorized data access. |
| Accounting | Accounting refers to logging and monitoring user and system activities to detect and respond to security incidents. Green Pace maintains detailed audit logs of login attempts, system modifications, and data access events. These logs are stored securely, monitored using a Security Information and Event Management (SIEM) system, and reviewed regularly to ensure compliance with security policies. Anomalous behavior triggers automated alerts, allowing for rapid investigation and response to potential threats. |

**\***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 | 02/15/2025 | Updated coding standards, risk assessments, and automation policies for DevSecOps compliance | Douglas Rowland | Douglas Rowland |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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