**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 | User input should never be trusted. Input data must be bounded and sanitized to prevent buffer overflow errors and SQL Injection attacks. Implement practices to verify input size and use both “allowlisting” and “denylisting” to manage acceptable input characters and sequences. |
| 1. Heed Compiler Warnings | Use the strictest compiler settings to catch potential errors. Ignoring warnings or reducing strictness can lead to undetected mistakes that might be exploited later. (OWASP Developer Guide | Principles of Security | OWASP Foundation, n.d.) |
| 1. Architect and Design for Security Policies | Use standardized architecture and design patterns to ensure code clarity and prevent errors. Collaborate across Development, Security, and Operations departments to integrate security into every phase. |
| 1. Keep It Simple | Complexity in code can introduce errors and make maintenance challenging. Prefer simpler solutions to avoid unnecessary complications. |
| 1. Default Deny | Processes should only proceed after completion of validation checks. |
| 1. Adhere to the Principle of Least Privilege | Grant only the necessary permissions required for actions, minimizing the potential for misuse. |
| 1. Sanitize Data Sent to Other Systems | Ensure data is sanitized before sending it to other systems to prevent the spread of potentially malicious code or errors. |
| 1. Practice Defense in Depth | Adopt DevSecOps practices and view security as a shared responsibility. |
| 1. Use Effective Quality Assurance Techniques | Employ static testing, unit testing, integration testing, penetration testing, and fuzzing to ensure code integrity and security. Explore and implement other testing tools and options to fill in any gaps discovered to create a comprehensive plan. Some degree of overlap between testing methods is encouraged and ensures security in different contexts. |
| 1. Adopt a Secure Coding Standard | Choose and adhere to secure coding standards appropriate for your organization's needs, ensuring consistent application of security practices. Refer to OWASP vulnerabilities to keep up with and adapt coding standards to modern trends and mitigate new vulnerabilities. |
| 1. Bonus – The Pareto Principle aka. The 80/20 rule. | The Pareto Principle, or the 80/20 rule, is a valuable philosophy in secure coding. It emphasizes focusing on the largest issues first, achieving the best results with the least effort. In secure coding, most issues, approximately 70%, stem from improperly handled input data. While it is important to not neglect other potential vulnerabilities, prioritizing the broad range of common issues should be the primary focus. This principle helps ensure that the most significant threats are mitigated efficiently, while still addressing less frequent but potentially severe vulnerabilities as needed.  The Pareto Principle is the principle I used to prioritize my choices for the secure coding practices. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | **Data Type Bounding** |

| **Noncompliant Code** |
| --- |
| **Description:** Using inappropriate or unsafe data types can lead to unexpected behavior or vulnerabilities. |
| int num;  num = 2147483648; // Out of range for an 'int' |

| **Compliant Code** |
| --- |
| **Description:** Use appropriate data types and ensure values are within acceptable ranges. |
| // Error handling catches the out of range int value before it can be used in a process  int num;  int value;  value = 2147483648; // Out of range for an 'int'  try {  if (value <= INT\_MAX) {  num = value;  throw std::out\_of\_range("This value is out of range");  }  }  catch (const std::out\_of\_range& e) {  std::cerr << "Caught out of range error: " << e.what() << "\n";  } |

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

| **Principles(s):**  **Validate Input Data:** Ensure all input data is bounded and sanitized to prevent out-of-range values and vulnerabilities.  **Keep It Simple:** Use clear and appropriate data types to avoid complex and error-prone code.  **Adhere to the Principle of Least Privilege:** Limit the range of data types to only what is necessary to avoid potential overflow or underflow errors.  **Practice Defense in Depth**: Implement checks for data type constraints to prevent unexpected behavior and ensure security.  These principles align with the Data Type Bounding standard by focusing on validation, simplicity, privilege, and layered defense. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | **Cppcheck** | | 2.14.2 | --enable=warning | Detects potential data type mismatches and range issues. |
| |  | | --- | |  |  |  | | --- | | **Clang-Tidy** | | 20.0.0 | |  | | --- | | cppcoreguidelines-narrowing-conversions |  |  | | --- | |  | | Flags narrowing Conversions that could lead to data loss or out-of-range errors. |
| |  | | --- | | **SonarQube** |  |  | | --- | |  | | 10.6 | cpp:S1479 | Identifies issues related to unsafe type conversions and potential overflow situations. |
| |  | | --- | | **PVS-Studio** |  |  | | --- | |  | | 7.32 | V501 | Detects potential integer overflow and out-of-range errors. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Data Value Validation** |

| **Noncompliant Code** |
| --- |
| **Description:** Failing to validate input values can lead to security issues like buffer overflows. |
| // bad no check  char buffer[10];  strcpy(buffer, userInput); // Unsafe copy  // better but also bad, prefer strncpy\_s to strncpy.  char buffer[10];  strncpy(buffer, userInput, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination |

| **Compliant Code** |
| --- |
| **Description:** Validate and sanitize input values to prevent overflows and other issues. |
| char buffer[10];  strncpy\_s(buffer, userInput, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination |

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

| **Principles(s):**  **Validate Input Data:** Always validate and sanitize input values to prevent security vulnerabilities such as buffer overflows.  **Default Deny:** Ensure that any data processed is validated before being used, preventing unsafe operations.  **Use Effective Quality Assurance Techniques:** Employ thorough testing, including boundary testing, to verify that data validation mechanisms are functioning correctly.  **Practice Defense in Depth:** Implement multiple layers of validation to safeguard against unexpected input and buffer overflow vulnerabilities.  These principles support the Data Value Validation standard by emphasizing input validation, default denial, quality assurance, and layered security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14.2 | --enable=warning | Detects unsafe usage of string functions like strcpy. |
| Clang-Tidy | 20.0.0 | cppcoreguidelines-pro-type-vararg | Flags potential issues with unsafe string handling and recommends safer alternatives. |
| SonarQube | 10.6 | cpp:S5482 | Identifies unsafe functions like strcpy and suggests safer alternatives. |
| PVS-Studio | 7.32 | V512 | Checks for possible buffer overflow issues and recommends safe string handling practices. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | **Use Safe String Handling** |

| **Noncompliant Code** |
| --- |
| **Description:** Incorrect handling of strings can lead to vulnerabilities such as buffer overflows or data corruption. |
| // Bad, No checking  char char\* str = "example";  char buffer[10];  strcpy(buffer, str); // Potential buffer overflow  // Better, but prefer strncopy\_s  const char \*str = "example";  char buffer[10];  strncpy(buffer, str, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Null-terminate |

| **Compliant Code** |
| --- |
| **Description:** Use safe string handling functions and verify buffer sizes. |
| const char\* str = "example";  char buffer[10];  strncpy\_s(buffer, str, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Null-terminate |

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

| **Principles(s):**  **Validate Input Data:** Ensure that string handling functions validate input to prevent buffer overflows and data corruption.  **Default Deny:** Use secure functions that check and limit the size of the data being processed to avoid buffer overflows.  **Keep It Simple:** Choose safer string handling methods that avoid complexity and potential vulnerabilities.  **Practice Defense in Depth**: Apply multiple safeguards, such as bounds checking and null-termination, to protect against string-related vulnerabilities.  These principles align with String Correctness by focusing on safe string handling, validation, simplicity, and layered protection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14.2 | --enable=warning | Detects unsafe string functions like strcpy. |
| Clang-Tidy | 20.0.0 | cppcoreguidelines-pro-bounds-array-to-pointer-decay | Flags unsafe string handling and array operations. |
| SonarQube | 10.6 | cpp:S5482 | Identifies and flags insecure string handling functions. |
| PVS-Studio | 7.32 | V512 | Checks for potential buffer overflow issues and recommends safe string practices. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **No Direct User Input in SQL Queries** |

| **Noncompliant Code** |
| --- |
| **Description:** Directly including user input in SQL queries can lead to SQL Injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| **Description:** Use parameterized queries or prepared statements to prevent SQL Injection. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  // Use parameterized query mechanism provided by the database library  // ‘?’ Represents the user input which can be pre-filtered before adding to the query |

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

| **Principles(s):**  **Validate Input Data -** Validating and sanitizing input prevents potential vulnerabilities such as buffer overflows and ensures data integrity.  **Keep It Simple -** Using straightforward and secure functions reduces the risk of errors and simplifies code maintenance.  **Sanitize Data Sent to Other Systems:** Properly sanitize and validate data before including it in SQL queries to avoid injecting malicious data into the database.  **Defense in Depth -** Implementing robust validation and sanitization practices adds an extra layer of protection against potential security issues.  These principles align with SQL Injection Prevention by focusing on input validation, parameterized queries, minimal permissions, and layered security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | Urgent | Critical |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **SonarQube** | 10.6 | cpp:S3649 | Detects and flags potential SQL injection vulnerabilities. |
| **PVS-Studio** | 7.32 | V5608 | Detects Possible SQL Injection due to potentially tainted data used to create SQL command. |
| **OWASP ZAP** | 2.15.0 | SQL Injection Scanner | Identifies and reports potential SQL injection points. |
| **Cppcheck** | 2.14.2 | Custom rules or plugins | Can be extended to check for unsafe SQL practices. |
| **Clang-Tidy** | 20.0.0 | Custom checks | Provides customizable checks that can be tailored for SQL. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **Prefer Smart Pointers** |

| **Noncompliant Code** |
| --- |
| **Description:** Ensures dynamically allocated memory is properly deallocated to prevent memory leaks. |
| // Noncompliant code example  void processData() {  int\* data = new int[100];  // Process data  // Memory leak: 'data' is not deleted  } |

| **Compliant Code** |
| --- |
| **Description:** Ensures dynamically allocated memory is properly deallocated to prevent memory leaks. |
| // Compliant code example  void processData() {  int\* data = new int[100];  // Process data  delete[] data; // Properly deallocate memory  }  // When possible, prefer smart pointers over manual pointers to simplify memory management.  std::unique\_ptr  std::weak\_ptr  std::shared\_ptr  std::auto\_ptr |

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

| **Principles(s):**  **Validate Input Data:** Ensure that memory allocations are handled safely and checked to prevent issues like memory leaks.  **Keep It Simple:** Use smart pointers to manage memory automatically and reduce the risk of leaks and errors associated with manual memory management.  **Practice Defense in Depth:** Implement multiple safeguards, such as proper memory deallocation and smart pointers, to protect against memory-related vulnerabilities.  **Adhere to the Principle of Least Privilege:** Limit the scope and lifetime of dynamically allocated memory to reduce potential misuse and errors.  These principles support Memory Protection by focusing on safe memory handling, simplicity, layered protection, and scoped resource management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18.1 | Memory Leak Detection | Detects memory leaks by tracking heap memory allocations. |
| Clang-Tidy | 20.0.0 | cppcoreguidelines-owning-memory | Warns about manual memory management issues. |
| PVS-Studio | 7.32 | V611 | Detects potential memory leaks and improper deallocation. |
| Cppcheck | 2.14.2 | memleak | Flags possible memory leaks in the code. |
| GCC AddressSanitizer | 13.1 | Memory Leak Checker | Runs runtime checks to detect memory management errors. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Assert Expected and Unexpected Results.** |

| **Noncompliant Code** |
| --- |
| **Description:** Not using assertions can lead to undetected logic errors or incorrect assumptions. |
| int x = getValue();  if (x < 0) {  // No assertion or error handling  …programming logic executes, and unexpected results occur.  } |

| **Compliant Code** |
| --- |
| **Description:** Use assertions to validate assumptions and catch errors during development. |
| int x = getValue();  assert(x >= 0); // Ensures x is non-negative |

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

| **Principles(s):**  **Validate Input Data:** Use assertions to validate assumptions and input values during development to catch errors early.  **Keep It Simple:** Implement assertions to ensure that code behaves as expected, making it easier to identify and fix logic errors.  **Use Effective Quality Assurance Techniques:** Employ assertions as a part of your quality assurance strategy to catch and address issues before deployment.  **Practice Defense in Depth:** Use assertions to add an extra layer of validation and catch potential errors or incorrect assumptions in the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.0.0 | cppcoreguidelines-assertions | Ensures assertions are used where appropriate. |
| Cppcheck | 2.14.2 | assert-side-effect | Detects misuse of assertions, including side effects in assert statements. |
| PVS-Studio | 7.32 | V590 | Warns about missing assertions or ineffective assertions. |
| GCC | 13.1 | Built-in | Supports the use of assertions in C++ code to catch logical errors. |
| Visual Studio Static Analyzer | 2023 | C6386 | Detects potential issues that could benefit from assertions. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | **Use Meaningful Exception handling** |

| **Noncompliant Code** |
| --- |
| **Description:** Improper handling of exceptions can lead to unhandled errors and application crashes. |
| try {  // Code that might throw an exception  } catch (...) {  // Generic catch-all with no handling  } |

| **Compliant Code** |
| --- |
| **Description:** Handle exceptions properly with specific catch blocks and meaningful error handling. |
| try {  // Code that might throw an exception  } catch (const std::exception &e) {  // Handle exception and log the error  } |

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

| **Principles(s):**  **Validate Input Data:** Ensure that exceptions related to invalid inputs are handled with specific catch blocks to address potential issues.  **Keep It Simple:** Use specific catch blocks to provide clear and meaningful error handling, simplifying the debugging and recovery process.  **Use Effective Quality Assurance Techniques:** Implement comprehensive exception handling as part of your quality assurance strategy to prevent unhandled errors and improve application stability.  **Practice Defense in Depth:** Employ specific and meaningful error handling to add an extra layer of protection against unhandled exceptions and application crashes.  These principles align with Exceptions by focusing on validation, simplicity, quality assurance, and layered error handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.0.0 | cppcoreguidelines-avoid-catch-all | Ensures that catch-all blocks are avoided in favor of specific exception types. |
| Cppcheck | 2.14.2 | catch-all | Detects the use of catch-all exception blocks and suggests specific handling. |
| PVS-Studio | 7.32 | V1228 | Warns about the usage of general catch blocks without meaningful handling. |
| GCC | 13.1 | Built-in | Supports detection of exception handling issues in C++ code. |
| Visual Studio Static Analyzer | 2023 | C6262 | Analyzes exception handling and suggests improvements. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Using Deprecated Functions | STD-008-CPP | Avoid Using Deprecated Functions |

| **Noncompliant Code** |
| --- |
| **Description:** Uses deprecated function gets which is unsafe and can lead to buffer overflows. |
| // Noncompliant code example  void readInput() {  char buffer[100];  gets(buffer); // Unsafe, deprecated function  } |

| **Compliant Code** |
| --- |
| **Description:** Uses ‘fgets’ to safely read input into a buffer with a specified size. |
| // Compliant code example  void readInput() {  char buffer[100];  fgets(buffer, sizeof(buffer), stdin); // Safe function with bounds checking  } |

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

| **Principles(s):**  **Validate Input Data:** Ensure that updated and safe functions are used to handle input, preventing potential buffer overflows and vulnerabilities.  **Keep It Simple:** Use current, secure functions to handle operations, avoiding deprecated and unsafe alternatives for simpler, more reliable code.  **Use Effective Quality Assurance Techniques:** Incorporate updated functions in testing to ensure code security and compliance with modern standards.  **Practice Defense in Depth:** Replace deprecated functions with secure alternatives to add an extra layer of protection against potential security risks and maintain code integrity.  These principles align with Avoid Using Deprecated Functions by focusing on validation, simplicity, quality assurance, and layered security through up-to-date practices |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.0.0 | cppcoreguidelines-avoid-deprecated-functions | Detects the use of deprecated functions and suggests modern alternatives. |
| Cppcheck | 2.14.2 | deprecatedFunction | Identifies and warns about the use of deprecated functions. |
| PVS-Studio | 7.32 | V1109 | Detects usage of deprecated and obsolete functions. |
| GCC | 13.1 | Built-in | Issues warnings when deprecated functions are used. |
| Visual Studio Static Analyzer | 2023 | C4996 | Warns about deprecated function usage and suggests safer alternatives. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Race Conditions | STD-009-CPP | Thread Safe Resource Access |

| **Noncompliant Code** |
| --- |
| **Description:** Uses shared resources in a multithreaded environment without proper synchronization, leading to race conditions. |
| // Noncompliant code example  int counter = 0;  void incrementCounter() {  counter++;  }  void decrementCounter() {  counter--;  }  // Called from multiple threads without synchronization |

| **Compliant Code** |
| --- |
| **Description:** Uses mutexes to synchronize access to shared resources, preventing race conditions. |
| // Compliant code example  int counter = 0;  std::mutex counterMutex;  void incrementCounter() {  std::lock\_guard<std::mutex> lock(counterMutex);  counter++;  }  void decrementCounter() {  std::lock\_guard<std::mutex> lock(counterMutex);  counter--;  }  // Safe to call from multiple threads with synchronization |

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

| **Principles(s):**    **Validate Input Data:** Ensure that access to shared resources is validated and synchronized to prevent race conditions and maintain data integrity.  **Keep It Simple:** Use straightforward synchronization mechanisms, such as mutexes, to manage access to shared resources and avoid complex error-prone solutions.  **Practice Defense in Depth:** Implement multiple layers of thread safety, such as mutexes and locks, to protect against race conditions and ensure consistent access to shared resources.  **Use Effective Quality Assurance Techniques:** Include multithreaded testing and analysis in your quality assurance process to identify and address potential race conditions.  These principles align with Avoid Race Conditions by focusing on synchronization, simplicity, layered protection, and thorough testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer (TSan) | Latest | Built-in | Dynamic analysis tool that detects data races and race conditions. |
| Clang-Tidy | Latest | cppcoreguidelines-avoid-race-condition | Identifies potential race conditions and suggests synchronization mechanisms. |
| PVS-Studio | Latest | V3147 | Detects possible race conditions in multithreaded code. |
| Cppcheck | Latest | raceCondition | Statically analyzes code for potential race conditions. |
| Visual Studio Static Analyzer | Latest | C26494 | Warns about uninitialized variables and potential race conditions. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Configuration Management | STD-010-CPP | No Hard Coded Data! |

| **Noncompliant Code** |
| --- |
| **Description:** Hardcodes sensitive configuration data such as passwords and API keys directly in the source code. |
| // Noncompliant code example  const char\* apiKey = "12345-ABCDE";  // Use apiKey for API requests |

| **Compliant Code** |
| --- |
| **Description:** Uses environment variables or secure configuration files to manage sensitive data. |
| // Compliant code example  const char\* apiKey = getenv("API\_KEY"); // Storing sensitive information in protected developer environments allows the keys to be retrieved and used without the need to hardcode values.  if (apiKey == nullptr) {  // Handle missing API key  } else {  // Use apiKey for API requests  } |

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

| **Principles(s):**  **Validate Input Data:** Ensure that sensitive configuration data is not hardcoded but securely retrieved from protected sources.  **Keep It Simple:** Use environment variables or secure configuration files to manage sensitive information, simplifying access and reducing risk.  **Adhere to the Principle of Least Privilege:** Limit access to sensitive data by storing it in secure locations and minimizing its exposure in the codebase.  **Practice Defense in Depth**: Implement multiple layers of security for managing sensitive information, including environment-based retrieval and secure configuration management.  These principles align with Secure Configuration Management by emphasizing secure handling, simplicity, minimal access, and layered security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Git Secrets | Latest | Secret Detection | Scans Git repositories for hardcoded secrets and sensitive information. |
| TruffleHog | Latest | Secret Scanning | Searches for high-entropy strings and potential secrets in codebases. |
| SonarQube | Latest | Security Plugin | Identifies hardcoded secrets and other security issues in the codebase. |
| Checkmarx | Latest | Static Application Security Testing (SAST) | Analyzes code for security vulnerabilities including hardcoded secrets. |
| Veracode | Latest | Static Analysis | Provides comprehensive security analysis including detection of hardcoded sensitive data. |

### 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 strengthen security and transition into a DevSecOps process and infrastructure, the first step is to foster collaborative decision-making among various departments. By integrating the tools and expertise of each specialization, the CI/CD pipeline can be seamlessly incorporated, minimizing disruptions and conflicts caused by incompatibilities. This integration also facilitates the automation of code reviews, testing, monitoring, and feedback processes. Modifying the CI pipeline to include static code analysis tools, such as SonarQube, enables automated code quality and security analysis, ensuring critical standards like Data Type Bounding and SQL Injection Prevention are enforced at the commit stage. During deployment, automated checks and security scans ensure compliance with standards like Memory Protection and String Correctness before code is released to production.

Automated code review tools and testing frameworks, such as CodeClimate and Google Test, can be integrated to uphold standards like Data Value Validation and Race Condition Avoidance. Continuous monitoring through SIEM systems and automated logging helps track adherence to security policies and address deviations promptly. Feedback mechanisms within the DevOps pipeline provide immediate responses to compliance issues, supporting ongoing improvement of both DevOps processes and coding standards. By embedding these automation strategies into the DevOps process, security practices become an integral part of development, spanning code creation, deployment, and monitoring.

### 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 | Medium | Low | High | Critical |
| STD-002-CPP | High | Medium | Medium | High | Critical |
| STD-003-CPP | High | Medium | Medium | High | Critical |
| STD-004-CPP | Critical | High | Low | Urgent | Critical |
| STD-005-CPP | High | Moderate | Moderate | High | Critical |
| STD-006-CPP | Medium | High | Low | High | Moderate |
| STD-007-CPP | High | Medium | Medium | High | Severe |
| STD-008-CPP | High | Medium | Low | High | Critical |
| STD-009-CPP | High | Medium | Medium | High | Critical |
| STD-010-CPP | High | Medium | Low | High | Critical |
| SYS-001 | High | Medium | High | High | Critical |
| SYS-002 | High | High | High | High | Critical |
| SYS-003 | High | Medium | High | High | Critical |
| SYS-004 | High | High | High | High | Critical |
| SYS-005 | High | High | High | High | Critical |
| SYS-006 | High | Medium | Medium | High | Critical |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 a policy for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to encrypting data while it is stored on a physical medium, ensuring its protection even when not actively accessed or processed. This can be achieved through technologies like Full Disk Encryption (FDE), which encrypts the entire storage drive, or File-Level Encryption, which secures individual files. This policy is essential for protecting sensitive information in storage, ensuring confidentiality and security even if the storage medium is compromised or accessed without authorization. |
| Encryption in flight | Encryption in flight (or in transit) encrypts data as it moves across networks, preventing interception and unauthorized access during transmission. This is typically done using protocols like TLS (Transport Layer Security) for web traffic or VPN (Virtual Private Network) protocols for secure remote connections. By encrypting data transmitted over public or insecure networks, confidentiality and integrity are maintained. This policy ensures data is protected from being intercepted and read by unauthorized parties during transmission, crucial for securing data exchanged between clients and servers, especially over public networks. |
| Encryption in use | Encryption in use involves encrypting data while it is actively processed or modified, ensuring its protection even when accessed. This can be achieved through technologies like Homomorphic Encryption, which enables computations on encrypted data without the need for decryption, or Secure Enclaves that safeguard data during its use. This approach is crucial for maintaining the confidentiality of sensitive information during processing, ensuring that data remains secure even when accessed by applications or users. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user or system before granting access to resources. Methods include passwords, two-factor authentication (MFA), single sign-on (SSO), and biometric verification, which confirm user identities before allowing access. This policy applies to all access points requiring user verification to prevent unauthenticated access of systems or sensitive data |
| Authorization | Authorization defines what a user is allowed to do after authentication, such as accessing specific resources or performing certain actions. It is managed through role-based access control (RBAC), attribute-based access control (ABAC), or policy-based access control, by setting permissions and access levels based on user roles or attributes. This policy ensures that after authentication, users have only the necessary access to use the system appropriately for their role. |
| Accounting | Accounting (or auditing) tracks and records user activities to ensure actions are logged for compliance and security review. This is achieved through logging and monitoring tools that capture all user actions in a system. The policy supports monitoring and reviewing user activity to detect unauthorized actions, ensure compliance, and aid investigations, maintaining system security and accountability. |

**\***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 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 always comply with this policy.

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 the 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 | 08/10/2024 | Filled out the Security Policy | Jeremiah Boothe | Jeremiah Boothe |
| 1.2 | [Insert text.] | [Insert text.] | Jeremiah Boothe | Jeremiah Boothe |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

# References

About secret scanning - GitHub Docs. (2024). GitHub Docs. <https://docs.github.com/en/code-security/secret-scanning/introduction/about-secret-scanning>

clang-tidy - Clang-Tidy Checks — Extra Clang Tools 20.0.0git documentation. (2024). Llvm.org. <https://clang.llvm.org/extra/clang-tidy/checks/list.html>

Jeganathan, S. (2019, November). *DevSecOps: A Systemic Approach for Secure Software Development*. Mydigitalpublication.com. <https://mydigitalpublication.com/publication/?i=632044&article_id=3524379&view=articleBrowser>

*OWASP Developer Guide | Principles of Security | OWASP Foundation*. (n.d.). Owasp.org. <https://owasp.org/www-project-developer-guide/draft/foundations/security_principles/>

PVS-Studio Messages. (2014). PVS-Studio. <https://pvs-studio.com/en/docs/warnings/>

Robert, S. (2024, February 14). SEI CERT C++ Coding Standard - SEI CERT C++ Coding Standard - Confluence. Wiki.sei.cmu.edu. <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

SAST. (n.d.). Checkmarx.com. <https://checkmarx.com/cxsast-source-code-scanning/>

Seacord, R. (2018). *Top 10 Secure Coding Practices - CERT Secure Coding - Confluence*. Cmu.edu. <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

Security ◆ Truffle Security Co. (2021). Trufflesecurity.com. <https://trufflesecurity.com/security?gclid=Cj0KCQjwt4a2BhD6ARIsALgH7Dpo7L-GRMZ3yCoE-eaVohZ3-KfUBsZ4KQ6U20JzR6oTL1omLYUdliAaAtzVEALw_wcB>

SonarQube 10.1. (n.d.). Docs.sonarsource.com. <https://docs.sonarsource.com/sonarqube/latest/>

The Most Powerful Application Security Platform on the Planet. (2016, November 14). Veracode. <https://www.veracode.com/>