**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

[C/C++ Ten Coding Standards 3](#_3dy6vkm)

[Coding Standard 1 4](#_1t3h5sf)

[Coding Standard 2 5](#_4d34og8)

[Coding Standard 3 6](#_2s8eyo1)

[Coding Standard 4 7](#_17dp8vu)

[Coding Standard 5 8](#_3rdcrjn)

[Coding Standard 6 9](#_26in1rg)

[Coding Standard 7 10](#_lnxbz9)

[Coding Standard 8 11](#_35nkun2)

[Coding Standard 9 13](#_1ksv4uv)

[Coding Standard 10 14](#_44sinio)

[Defense-in-Depth Illustration 15](#_2jxsxqh)

[Project One 15](#_z337ya)

[1.](#_3j2qqm3) Revise the C/C++ Standards 15

[2.](#_1y810tw) Risk Assessment 15

[3.](#_4i7ojhp) Automated Detection 15

[4.](#_2xcytpi) Automation 15

[5.](#_1ci93xb) Summary of Risk Assessments 16

[6.](#_3whwml4) Create Policies for Encryption and Triple A 16

[7.](#_2bn6wsx) Map the Principles 17

[Audit Controls and Management 18](#_qsh70q)

[Enforcement 18](#_3as4poj)

[Exceptions Process 18](#_1pxezwc)

[Distribution 19](#_49x2ik5)

[Policy Change Control 19](#_2p2csry)

[Policy Version History 19](#_147n2zr)

[Appendix A Lookups 19](#_3o7alnk)

[Approved C/C++ Language Acronyms 19](#_23ckvvd)

## 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 | Developers should validate all input data, trusted or untrusted. Ensuring that the data conforms to expected formats and values can greatly minimize the risk of vulnerabilities. Proper validation helps in preempting the introduction of malicious data into the system. |
| 1. Heed Compiler Warnings | Pay attention to all compiler warnings. While some warnings might appear minor, they can often highlight potential security issues in the codebase. Addressing these warnings promptly can help ensure the safety of the application. |
| 1. Architect and Design for Security Policies | Be proactive in incorporating security considerations, developers can both optimize the functionality of the code and safeguard against potential vulnerabilities ahead of time, leading to a more resilient final product. |
| 1. Keep It Simple | Clean and well-organized code is not only easier to maintain and understand but also significantly reduces the attack surface for potential adversaries. A simple codebase can be more readily audited and protected against security threats. |
| 1. Default Deny | Deny all requests or actions unless they've been expressly permitted. This approach ensures that only verified activities are allowed, reducing the potential for unexpected or malicious behavior. |
| 1. Adhere to the Principle of Least Privilege | Ensure that every module, process, or user operates with the minimal set of privileges necessary to complete its function. Limiting access rights and permissions can substantially reduce the impact of a potential security breach or system misuse. |
| 1. Sanitize Data Sent to Other Systems | It is essential to sanitize and cleanse data before sending it to other systems or components. Developers should strip or neutralize any data that could be interpreted or executed, minimizing the risk of cross-system. |
| 1. Practice Defense in Depth | Security should be approached in layers. Implement multiple levels of security controls and countermeasures. If one defense mechanism fails, another should be in place to catch the problem, ensuring more comprehensive security. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance processes are foundational to secure software development. Developers should employ testing, code reviews, and other QA techniques to identify potential vulnerabilities or flaws in the codebase. |
| 1. Adopt a Secure Coding Standard | By following a standard, developers can avoid known pitfalls and vulnerabilities, benefiting from best practices established by collective experience. Regularly updating their knowledge based on these standards will also ensure that they stay abreast of emerging threats and mitigation techniques. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Proper Use of Data Types: Avoid using incorrect data types (such as int for bool). |

| **Noncompliant Code** |
| --- |
| An int variable is used to represent a bool value, leading to potential confusion. |
| int isActive = 1; // Using int to represent boolean value  if (isActive) {  // Do something  } |

| **Compliant Code** |
| --- |
| The bool data type is correctly used to represent a boo value, improving readability. |
| bool isActive = true; // Using bool for clarity  if (isActive) {  // Do something  } |

| **Principles(s):** Validate Input Data; By ensuring only the correct data types are used, the standard enacts the principle of validating input data, preventing potential vulnerabilities from improperly processed data.  Keep It Simple; Using the proper data types streamlines the codebase, embodying the principle of keeping the code simple and reducing potential points of confusion or exploitation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | useBool | Suggests using bool for boolean values. |
| Clang-Tidy | 13.0.0 | misc-bool-pointer-implicit-conversion | Warns on implicit conversions from integer to bool. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Initialization of Data Values: Ensure that data values are initialized before use to prevent undefined behavior. |

| **Noncompliant Code** |
| --- |
| The variable value is used without being initialized. |
| int value; // Uninitialized variable  int result = value + 5; |

| **Compliant Code** |
| --- |
| The variable value is correctly initialized before use. |
| int value = 0; // Initialized variable  int result = value + 5; |

| **Principles(s):** Validate Input Data; Ensuring data values are initialized before use aligns with the principle of validating input data, which helps in reducing vulnerabilities by ensuring data adheres to expected formats and values.  Keep It Simple; By initializing data values, the code becomes less prone to unexpected errors, embodying the principle of keeping the code simple and easily auditable against security threats. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | uninitvar | Check for uninitialized variables |
| Clang | 13.0.0 | alpha.core.init | Detects uninitialized variables. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP | Safe String Handling: Ensure strings are properly terminated and bounds-checked to prevent overflows and undefined behavior. |

| **Noncompliant Code** |
| --- |
| The string "hello" does not fit into str when considering the null terminator. |
| char str[5];  strcpy(str, "hello"); |

| **Compliant Code** |
| --- |
| The string "hello" correctly fits into str including the null terminator. |
| char str[6];  strcpy(str, "hello"); |

| **Principles(s):** Adhere to the Principle of Least Privilege; By ensuring strings are bounds-checked, the code only accesses the memory it's supposed to.  Practice Defense in Depth; Safe string handling is a layer in the multi-faceted approach to security, ensuring that even if other defenses fail, string-related vulnerabilities do not become an issue. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | bufferAccessOutOfBounds | Detects out-of-bounds accesses |
| Coverity | 2020.06 | buffer-overrun | Warns on potential array OOB |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent Injection: Avoid using string concatenation for SQL queries to prevent vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Uses direct string concatenation for building the SQL query. |
| char \*userInput = ...; // Obtained from user  char query[256];  sprintf(query, "SELECT \* FROM users WHERE name='%s'", userInput);  execute(query); |

| **Compliant Code** |
| --- |
| Safely prepares and binds the user input to the SQL query. |
| char \*userInput = ...; // Obtained from user  char query[] = "SELECT \* FROM users WHERE name=?";  prepare(query);  bindParameter(1, userInput);  execute(); |

| **Principles(s):** Validate Input Data; Using parameterized SQL queries or prepared statements instead of string concatenation ensures that the data is treated as input rather than executable code.  Adopt a Secure Coding Standard; By adhering to standards that prevent SQL injection, developers incorporate collective best practices to avoid commonly known vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2020.06 | SQLI | Detects potential SQL Injection Vulnerabilities |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Safe Dynamic Memory Management: Use smart pointers in C++ to manage dynamically allocated memory and prevent memory leaks or undefined behavior. |

| **Noncompliant Code** |
| --- |
| Accesses memory after it has been deleted, causing undefined behavior. |
| int\* ptr = new int[5];  delete[] ptr;  ptr[3] = 42; |

| **Compliant Code** |
| --- |
| Uses unique\_ptr to manage the dynamic memory. |
| #include <memory>  std::unique\_ptr<int[]> ptr(new int[5]);  ptr[3] = 42; // Safe memory access |

| **Principles(s):** Keep It Simple; Utilizing smart pointers simplifies memory management by automating deallocation, reducing the potential for human error and ensuring safer code.  Adopt a Secure Coding Standard; Implementing memory management best practices aligns with secure coding guidelines, reducing vulnerabilities associated with manual memory management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | dealloc | Warns about deallocating an already deallocated pointer |
| clang | 13.0.0 | use-after-free | Detects use of memory after it has been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Over-use of assertions: only use assertions to verify invariants, not for handling conditions or errors. |

| **Noncompliant Code** |
| --- |
| The function improperly uses an assertion to check an input condition. |
| #include <cassert>  void processInput(int value) {  assert(value >= 0); // Relying on assertions for external input  //…  } |

| **Compliant Code** |
| --- |
| The function correctly checks the condition without relying on assertions. |
| void processInput(int value) {  if (value < 0) {  // Handle error or return  }  // ... other code ...  } |

| **Principles(s):** Architect and Design for Security Policies; Using assertions appropriately aligns with designing code that is both functional and secure, ensuring that assertions arent being misused for error handling.  Keep It Simple; Restricting the use of assertions to their intended purpose results in cleaner and more maintainable code, preventing potential misinterpretations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | assertWithSideEffect | Warns when an assertion is used with a function that has side effects |
| Clang | 13.0.0 | improper-assert | Identifies the use of assertions in non-invariant scenarios |

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Safe Exception Handling: ensure exceptions are caught and handled appropriately without ignoring errors. |

| **Noncompliant Code** |
| --- |
| Catches all exceptions indiscriminately and does not handle or log them. |
| void riskyFunction() {  try {  // some throw  } catch (...) {  // Catch all exceptions and do nothing  }  } |

| **Compliant Code** |
| --- |
| Catches specific exceptions and provides an opportunity to handle them. |
| void safeFunction() {  try {  // some throw  } catch (const std::exception& e) {  // Handle known exception type  }  } |

| **Principles(s):** Use Effective Quality Assurance Techniques; Proper exception handling is a key aspect of quality assurance, ensuring that software behaves predictably. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | catchAll | Warns about catching general exceptions without handling |
| Clang | 13.0.0 | catch-all-exception | Warns on catch all without proper handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Inheritance | STD-008-CPP | Safe Inheritance Practices: Ensure base classes are designed to be inherited and prevent unintended behaviors when subclassing. |

| **Noncompliant Code** |
| --- |
| The Derived class silently overrides the doWork method, which might not be intended by the Base class design. |
| class Base {  public:  void doWork() {  // Implementation  }  };  class Derived : public Base {  public:  void doWork() {  // Different implementation  }  }; |

| **Compliant Code** |
| --- |
| Base class explicitly declares doWork as virtual, signaling it's designed to be overridden. The Derived class uses the override specifier to ensure it's intentionally overriding the method. A virtual destructor is added for safe destruction. |
| class Base {  public:  virtual void doWork() {  // Implementation  }  virtual ~Base() = default; // Virtual destructor  };  class Derived : public Base {  public:  void doWork() override {  // Different implementation  }  }; |

| **Principles(s):** Architect and Design for Security Policies; When designing base classes with safe inheritance in mind you are actively preventing potential vulnerabilities due to unintended subclass behaviors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | checkOverride | Checks if override hides base class functions |
| Clang | 13.0.0 | -Winconsistent-missing-override | Warns when the override keyword is not applied in a derived class. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integer Overflow | STD-009-CPP | Guard Against Integer Overflow: Ensure arithmetic operations on integers don't result in overflow or underflow. |

| **Noncompliant Code** |
| --- |
| Adding 1 to the maximum integer value results in an overflow. |
| int maxValue = INT\_MAX;  int result = maxValue + 1; |

| **Compliant Code** |
| --- |
| Adding 1 to the maximum integer value results in an overflow. |
| #include <climits>  int maxValue = INT\_MAX;  if (maxValue < INT\_MAX) {  int result = maxValue + 1;  } |

| **Principles(s):**Validate Input Data; By checking the values involved in arithmetic operations, we can ensure they are within expected ranges, avoiding potential overflow or underflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.5 | arithmeticException | Warns about integer under/overflows |
| Clang | 13.0.0 | -Winteger-overflow | Warns about integer under/overflows |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Floating points | STD-010-CPP | Precision in Floating-Point Comparisons: account for possible precision errors. |

| **Noncompliant Code** |
| --- |
| Directly compares floating-point values, which might not yield expected results due to precision errors in representation. |
| double value = 0.1 + 0.2;  if (value == 0.3) {  //…  } |

| **Compliant Code** |
| --- |
| Uses the C++ standard library function std::abs and a tolerance range for floating-point comparison to account for potential precision errors. |
| #include <cmath>  double value = 0.1 + 0.2;  if (std::abs(value - 0.3) < 1e-9) {  //…  } |

| **Principles(s):** Use Effective Quality Assurance Techniques; Ensuring accurate floating-point comparisons is crucial for the overall reliability of the project, it is essential to implement a validation process or tests for floating point comparisons. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 13.0.0 | -Wfloat-equal | Warns when floating-point numbers are compared directly |

Defense-in-Depth Illustration



DevSecOps Diagram



**Automation**

In the DevSecOps pipeline, during the pre-production build stage, code should pass static checks by Cppcheck and the Clang static analyzer at commit time. These tools will perform static code analysis, helping to identify vulnerabilities before a manual code review even takes place. Integrating these scans into this phase ensures that any detected issues can be fixed before code gets pushed to the main repo. During the production health check phase, Coverity can be used to perform final static analysis, since it covers more languages and could help identify SQL injection or dangerous language interactions.

### 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 | Low | Likely | Low | Medium | 3 |
| STD-002-CPP | Low | Likely | Medium | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | High | Very Likely | High | Critical | 5 |
| STD-005-CPP | High | Likely | Low | High | 4 |
| STD-006-CPP | Medium | Likely | Low | Medium | 3 |
| STD-007-CPP | Medium | Likely | Low | Medium | 3 |
| STD-008-CPP | Medium | Possible | Low | Medium | 3 |
| STD-009-CPP | High | Very Likely | Medium | High | 4 |
| STD-010-CPP | Medium | Likely | Low | Medium | 2 |

### Policies for Encryption and Triple A

| **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | All stored sensitive data must be encrypted, regardless of the medium. This protects data from unauthorized access if storage mediums are compromised. |
| Encryption at flight | All data transmitted over public networks must be encrypted using industry-recognized protocols. This ensures data remains confidential during transmission and reduces the risk of man-in-the-middle attacks. |
| Encryption in use | Sensitive data loaded into memory for processing should be encrypted or obfuscated. This safeguards data from memory dump attacks and unauthorized in-system access. |

| **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users must provide valid credentials before access, including during user logins and when adding new users. This guarantees that only verified individuals can access resources, upholding system integrity. |
| Authorization | Access is based on the least privilege principle. Permissions, including changes to the database, user level of access, and files accessed, must be granted explicitly. This limits potential damage by containing what a compromised account can access or modify. |
| Accounting | All actions, such as user logins, changes to the database, new user additions, user access levels, and files accessed, must be logged and maintained for a set period. This logging enhances accountability and supports potential investigations. |

## 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 | 09/15/2023 | Initial Template | Dante Trisciuzzi |  |
| 1.1 | 10/03/2023 | Project One | Dante Trisciuzzi |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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