**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 | Ensure all inputs are validated to prevent attacks such as SQL injection, buffer overflow, or cross-site scripting. |
| 1. Heed Compiler Warnings | Pay attention to all warnings from the compiler to identify potential vulnerabilities. |
| 1. Architect and Design for Security Policies | Incorporate security policies into the design and architecture phase. |
| 1. Keep It Simple | Simplify code to make it more secure and easier to maintain. |
| 1. Default Deny | Establish "deny by default" rules to minimize unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Provide only the minimum permissions required to users or systems. |
| 1. Sanitize Data Sent to Other Systems | Prevent data injection or malicious manipulation by sanitizing outgoing data. |
| 1. Practice Defense in Depth | Use multiple layers of security controls to mitigate risks. |
| 1. Use Effective Quality Assurance Techniques | Apply QA processes like static code analysis and automated testing to catch vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Implement standardized secure coding practices to reduce risks. |

### 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 | Enforce adherence to the one-definition rule to maintain consistent type definitions. |

| **Noncompliant Code** |
| --- |
| This example shows two translation units defining the same structure with conflicting definitions. This inconsistency can lead to undefined behavior during compilation or runtime. |
| // a.cpp  struct S { int a; };  // b.cpp  class S { public: int a; }; |

| **Compliant Code** |
| --- |
| Using a common header file ensures consistent type definitions across all translation units, reducing potential errors and ensuring adherence to the one-definition rule |
| [// S.h  struct S { int a; };  // a.cpp  #include "S.h"  // b.cpp  #include "S.h" |

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

| **Principles(s):** Validate Input Data: Ensures consistent data definitions and safe integration.  Adopt a Secure Coding Standard: Maintains consistent and reliable code across the project. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 14.0 | One-Definition Rule | Identifies multiple conflicting definitions. |
| |  | | --- | | GCC | |  |  |  | | --- | |  | | 11.2 | ODR Warnings | Flags ODR violations in code |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Prevent Usage of Uninitialized Variables |

| **Noncompliant Code** |
| --- |
| Variables used without initialization can produce unpredictable results or undefined behavior. |
| void f() {  int x;  std::cout << x;  } |

| **Compliant Code** |
| --- |
| Initialize variables before use. |
| void f() {  int x = 0;  std::cout << x;  } |

| **Principles(s):** Heed Compiler Warnings: Warns developers of uninitialized variables.  Use Effective Quality Assurance Techniques: Helps catch these issues during static analysis. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | |  | | --- | | 3.19 |  |  | | --- | |  | | |  | | --- | | Memcheck |  |  | | --- | |  | | |  | | --- | | Detects usage of uninitialized memory. |  |  | | --- | |  | |
| |  | | --- | | Clang Tidy |  |  | | --- | |  | | |  | | --- | | 14.0 |  |  | | --- | |  | | |  | | --- | | Uninitialized Var |  |  | | --- | |  | | |  | | --- | | Flags uninitialized variables. | |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Avoid Creating Strings From Null Pointers |

| **Noncompliant Code** |
| --- |
| Constructing a string from a null pointer results in undefined behavior. |
| std::string s = std::getenv("NON\_EXISTENT\_VAR"); |

| **Compliant Code** |
| --- |
| Check for null before constructing the string. |
| const char\* envVar = std::getenv("NON\_EXISTENT\_VAR");  std::string s = envVar ? envVar : ""; |

| **Principles(s):** **Sanitize Data Sent to Other Systems: Ensures all data is valid and safe for downstream systems.**  **Keep It Simple: Avoids unexpected behavior by handling null safely.** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | SonarQube |  |  | | --- | |  | | |  | | --- | | 9.9 |  |  | | --- | |  | | Null Dereference | Detects potential null pointer dereferences. |
| Clang Static Analyzer | |  | | --- | | 14.0 |  |  | | --- | |  | | |  | | --- | | Null Analysis |  |  | | --- | |  | | Flags possible null pointer dereference errors. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection by Using Parameterized Queries. |

| **Noncompliant Code** |
| --- |
| Direct concatenation of user input into SQL queries enables attackers to inject malicious SQL commands, compromising database integrity. |
| std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "'"; database.execute(query); |

| **Compliant Code** |
| --- |
| Use parameterized queries or prepared statements. |
| PreparedStatement stmt = database.prepare("SELECT \* FROM users WHERE username = ?"); stmt.bind(1, userInput); stmt.execute(); |

| **Principles(s):** **Validate Input Data**: Ensures input meets predefined formats and prevents injection.   **Sanitize Data Sent to Other Systems**: Safeguards interactions with external databases. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | SQL Injection Rule | Detects potential SQL injection Vulnerabilities. |
| OWASP ZAP | 2.11 | SQL Injection Scanner | Scans for SQL injection risks in queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Avoid Accessing Freed Memory |

| **Noncompliant Code** |
| --- |
| Accessing memory after it has been freed results in undefined behavior.  cpp |
| int\* ptr = new int(10); delete ptr; std::cout << \*ptr; // Undefined behavior |

| **Compliant Code** |
| --- |
| Ensure pointers are nullified after memory is deallocated. |
| int\* ptr = new int(10);  delete ptr;  ptr = nullptr;  if (ptr) {  std::cout << \*ptr;  } |

| **Principles(s):** **Practice Defense in Depth**: Adds redundancy to prevent memory corruption.  **Use Effective Quality Assurance Techniques**: Identifies these risks during testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.19 | Memcheck | Detects access to freed memory |
| Clang Static Analyzer | 14.0 | Dangling Pointer Rule | Identifies use-after-free scenarios |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use Static Assertions for Compile-Time Errors |

| **Noncompliant Code** |
| --- |
| Runtime assertions should not be used for conditions that can be validated at compile time |
| #include <cassert> assert(sizeof(int) == 4); |

| **Compliant Code** |
| --- |
| Static assertions prevent code compilation if the condition fails, ensuring correctness early in the build process. |
| static\_assert(sizeof(int) == 4, "Unexpected int size"); |

| **Principles(s):** **Heed Compiler Warnings**: Catches issues early during compilation.  **Keep It Simple**: Reduces runtime checks by resolving issues at compile time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC | 11.2 | Static Assertion Checker | Ensures static assertions are used appropriately |
| Clang Tidy | 14.9 | Static Assert Rule | Flags incorrect or missing static assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Do Not Abruptly Terminate Programs on Exceptions |

| **Noncompliant Code** |
| --- |
| Unhandled exceptions lead to unexpected termination, risking data loss or corruption. |
| void riskyFunction() { throw std::runtime\_error("Error occurred"); } riskyFunction(); // Program terminates |

| **Compliant Code** |
| --- |
| Exception handling ensures graceful degradation and recovery from errors. |
| try { riskyFunction(); } catch (const std::exception& e) { std::cerr << "Exception: " << e.what() << std::endl; } |

| **Principles(s):**  **Practice Defense in Depth**: Adds layers of error management to protect against failures.  **Use Effective Quality Assurance Techniques**: Verifies proper handling of exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 14.0 | Exception Handling | Detects unhandled exceptions |
| |  | | --- | | SonarQube |  |  | | --- | |  | | 9.9 | Exception Management | Highlights unhandled or abrupt exception usage. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-008-CPP | Ensure Proper Cleanup of Resources |

| **Noncompliant Code** |
| --- |
| Resources like file handles or memory allocations are not released, leading to resource leaks. |
| FILE\* file = fopen("data.txt", "r"); // Exception occurs, file remains open |

| **Compliant Code** |
| --- |
| RAII ensures resources are released when no longer needed, even during exceptions. |
| std::ifstream file("data.txt");  // Automatically closed when out of scope |

| **Principles(s):** Adhere to the Principle of Least Privilege: Restricts resource access to prevent misuse.  Keep It Simple: Automatically handles resource cleanup, reducing errors. [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.19 | Resource Leak Checker | Detects memory and resource leaks. |
| Clang Static Analyzer | 14.0 | RAII Usage | Flags improper resource management. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Iterator Safety | STD-009-CPP | Ensure Valid Iterator Ranges |

| **Noncompliant Code** |
| --- |
| Passing invalid iterator ranges causes undefined behavior, as the end iterator precedes the begin iterator in this example. |
| #include <algorithm> #include <vector> std::vector<int> vec = {1, 2, 3}; std::for\_each(vec.end(), vec.begin(), [](int n) { std::cout << n; }); |

| **Compliant Code** |
| --- |
| Ensures that iterators are used in a valid range, maintaining predictable behavior. |
| #include <algorithm> #include <vector> std::vector<int> vec = {1, 2, 3}; std::for\_each(vec.begin(), vec.end(), [](int n) { std::cout << n; }); |

| **Principles(s):** **Validate Input Data**: Confirms that iterator ranges are valid before use.  **Practice Defense in Depth**: Adds safeguards against common iterator misuse. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Tidy | 14.0 | Iterator Safety Check | Identifies invalid iterator ranges. |
| GCC | 11.2 | Range Validation Checker | Ensures iterators are used in valid ranges |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Constructor Initialization | STD-010-CPP | Initialize Members in Declaration Order |

| **Noncompliant Code** |
| --- |
| Members are initialized in the order they are declared in the class, not in the initializer list order. This mismatch can lead to subtle bugs. |
| class MyClass {  int x;  int y;  public:  MyClass(int a, int b) : x(a), y(b) {}  }; |

| **Compliant Code** |
| --- |
| Aligning the initializer list with the declaration order ensures predictable and bug-free initialization. |
| class MyClass { int x; int y; public: MyClass(int a, int b) : x(a), y(b) {} }; |

| **Principles(s):** **Architect and Design for Security Policies**: Encourages structured and predictable code.  **Keep It Simple**: Simplifies understanding and debugging of initialization logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Tidy | 14.0 | Constructor Initialization | Detects incorrect initialization order. |
| SonarQube | 9.9 | Initialization Rules | |  | | --- | |  |  |  | | --- | | Highlights initializer list issues. | |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation tools will be implemented at various stages of the DevSecOps pipeline to enforce coding standards and detect vulnerabilities early. Tools like SonarQube and Clang Static Analyzer will be integrated during the verify phase for static analysis. Runtime tools like Valgrind will be utilized during pre-production testing to identify memory and resource issues. The release phase will incorporate automation to verify software integrity and enforce security measures such as code signing.

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

### 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 | Policy: All sensitive data stored within the Green Pace systems must be encrypted using AES-256 encryption.  Use: Encryption at rest ensures that data stored in databases, file systems, or other storage solutions is protected from unauthorized access, even if physical media is compromised.  Application: Encrypt database entries, backups, and files stored in servers or cloud storage. Access controls and encryption keys must be securely managed using a Key Management System (KMS).  Why: Protects sensitive information from theft or unauthorized access due to server breaches or physical theft. |
| Encryption in flight | **Policy**: All data transmitted between systems, users, or third-party applications must use secure transport protocols such as TLS 1.3.  **Use**: Encryption in flight protects data integrity and confidentiality during transmission over potentially insecure networks.  **Application**: All APIs, websites, and file transfers must use HTTPS, secure FTP, or other encrypted communication channels. Use client and server certificates for additional authentication layers.  **Why**: Ensures sensitive information like login credentials and transaction details cannot be intercepted during transmission. |
| Encryption in use | **Policy**: Implement memory encryption using secure enclaves or technologies like Intel SGX or AMD SEV to protect data actively being processed.  **Use**: Encryption in use protects sensitive data when it is being actively used in memory or CPU registers.  **Application**: Employ encrypted workloads in cloud environments and enable secure enclaves for handling sensitive computations.  **Why**: Mitigates risks of memory dumps, side-channel attacks, or unauthorized access to in-memory data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Policy**: All systems and applications must implement multi-factor authentication (MFA) for user logins.  **Use**: Authentication ensures that only authorized individuals can access the system.  **Application**: Use password policies with strong requirements, implement MFA using authenticator apps or hardware tokens, and periodically review authentication logs.  **Why**: Prevents unauthorized access due to compromised credentials, ensuring the identity of users accessing systems. |
| Authorization | **Policy**: Role-based access control (RBAC) must be implemented to ensure users only access resources necessary for their role.  **Use**: Authorization ensures users and systems can only perform permitted actions.  **Application**: Define clear roles with minimal privileges, audit access permissions quarterly, and ensure dynamic role reassignment for personnel changes.  **Why**: Reduces risk of privilege abuse and limits the impact of compromised accounts. |
| Accounting | **Policy**: Maintain logs of all user activities, including logins, database changes, file access, and administrative actions. Logs must be securely stored and regularly reviewed.  **Use**: Accounting ensures traceability and accountability for all system actions.  **Application**: Implement centralized logging systems like ELK or Splunk, ensure logs are encrypted, and define alert thresholds for suspicious activities.  **Why**: Enables forensic analysis, compliance with regulations, and monitoring for security breaches. |

**\***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.

|  |  |  |  |
| --- | --- | --- | --- |
| Standard Number | Standard | Principle(s) | Justification |
| STD-001-CPP | Validate Input Data | 1. Validate Input Data, 7. Sanitize Data Sent to Other Systems | Validation and sanitization ensure input is safe and conforms to expectations, reducing vulnerabilities like SQL injection, buffer overflows, and cross-site scripting. |
| STD-002-CPP | Heed Compiler Warnings | 2. Heed Compiler Warnings, 9. Use Effective Quality Assurance Techniques | Compiler warnings highlight potential security flaws, and addressing them ensures robust and reliable code. QA ensures the warnings are not ignored. |
| STD-003-CPP | Architect and Design for Security Policies | 3. Architect and Design for Security Policies, 8. Practice Defense in Depth | Security must be planned during architecture to build systems resistant to attacks. Defense in depth ensures layered protections are integrated into the architecture. |
| STD-004-CPP | Keep It Simple | 4. Keep It Simple, 6. Adhere to the Principle of Least Privilege | Simplicity reduces code complexity, making vulnerabilities easier to spot. The least privilege principle limits the impact of compromised components. |
| STD-005-CPP | Default Deny | 5. Default Deny, 8. Practice Defense in Depth | Denying access by default reduces the attack surface. Defense in depth adds additional layers, ensuring unauthorized access remains blocked even if one layer is bypassed. |
| STD-006-CPP | Adhere to the Principle of Least Privilege | 6. Adhere to the Principle of Least Privilege, 8. Practice Defense in Depth | Restricting permissions ensures users and processes only access what they need. Combining this with layered defenses minimizes potential attack vectors. |
| STD-007-CPP | Sanitize Data Sent to Other Systems | 1. Validate Input Data, 7. Sanitize Data Sent to Other Systems | Validating and sanitizing data protects systems by ensuring data integrity and mitigating risks of injection or code execution attacks. |
| STD-008-CPP | Practice Defense in Depth | 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques | Multiple layers of protection ensure that if one mechanism fails, others remain effective. QA processes verify the effectiveness of these layers. |
| STD-009-CPP | Use Effective Quality Assurance Techniques | 9. Use Effective Quality Assurance Techniques, 2. Heed Compiler Warnings | Thorough QA processes, including code reviews and automated testing, identify vulnerabilities early. Compiler warnings complement QA by flagging potential issues. |
| STD-010-CPP | Adopt a Secure Coding Standard | 10. Adopt a Secure Coding Standard, 3. Architect and Design for Security Policies | Secure coding standards enforce best practices, creating consistent and secure codebases. Designing with security ensures these standards are applied during initial development. |

**Validate Input Data:**

Mapped to Validate Input Data and Sanitize Data Sent to Other Systems.

Inputs are validated and sanitized to ensure that systems only process safe, expected data, minimizing risks of injection or malformed inputs.

Heed Compiler Warnings:

Mapped to Heed Compiler Warnings and Use Effective Quality Assurance Techniques.

Compiler warnings point out flaws that could be exploited. QA ensures developers do not miss these critical warning signals.

Architect and Design for Security Policies:

Mapped to Architect and Design for Security Policies and Practice Defense in Depth.

By embedding security into the design phase, vulnerabilities are prevented before implementation. Layered security strengthens overall defenses.

Keep It Simple:

Mapped to Keep It Simple and Adhere to the Principle of Least Privilege.

Simple designs reduce complexity and the likelihood of bugs. Limiting access ensures no unnecessary permissions are granted.

Default Deny:

Mapped to Default Deny and Practice Defense in Depth.

Blocking access by default ensures unauthorized access attempts are stopped. Layered defenses add redundancy against breaches.

Adhere to the Principle of Least Privilege:

Mapped to Adhere to the Principle of Least Privilege and Practice Defense in Depth.

Minimal access rights reduce the attack surface. Layered protections ensure that breached privileges have limited impact.

Sanitize Data Sent to Other Systems:

Mapped to Validate Input Data and Sanitize Data Sent to Other Systems.

Data integrity is protected by validating and sanitizing inputs before they reach their destination.

Practice Defense in Depth:

Mapped to Practice Defense in Depth and Use Effective Quality Assurance Techniques.

Multiple layers provide redundancy. QA confirms that these measures are properly implemented and effective.

Use Effective Quality Assurance Techniques:

Mapped to Use Effective Quality Assurance Techniques and Heed Compiler Warnings.

QA methods identify and prevent flaws. Compiler warnings complement QA by highlighting risks that automated testing may overlook.

Adopt a Secure Coding Standard:

Mapped to Adopt a Secure Coding Standard and Architect and Design for Security Policies.

Coding standards ensure consistent application of security practices. Secure design principles ensure these standards are enforced systematically**.**

Each standard is aligned with the principles that ensure secure practices are applied effectively in Green Pace’s software development processes. By linking specific principles to standards, the security policy demonstrates a clear and systematic approach to implementing security across the organization's development lifecycle.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 11/05/2024 | Milestone 1 Completion | Justin Dougherty |  |
| 3.0 | 12/08/2024 | Project One Completion | Justin Dougherty |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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