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



# Green Pace Secure Development Policy

## Contents

[Overview](#bookmark) 2

[Purpose](#bookmark1) 2

[Scope](#bookmark2) 2

[Module Three Milestone](#bookmark3) 2

[Ten Core Security Principles](#bookmark4) 2

[C/C++ Ten Coding Standards](#bookmark5) 3

[Coding Standard 1](#bookmark6) 4

[Coding Standard 2](#bookmark7) 5

[Coding Standard 3](#bookmark8) 6

[Coding Standard 4](#bookmark9) 7

[Coding Standard 5](#bookmark10) 8

[Coding Standard 6](#bookmark11) 9

[Coding Standard 7](#bookmark12) 10

[Coding Standard 8](#bookmark13) 11

[Coding Standard 9](#bookmark14) 13

[Coding Standard 10](#bookmark15) 14

[Defense-in-Depth Illustration](#bookmark16) 15

[Project One](#bookmark17) 15

[1. Revise the C/C++ Standards](#bookmark18) 15

[2. Risk Assessment](#bookmark19) 15

[3. Automated Detection](#bookmark20) 15

[4. Automation](#bookmark21) 15

[5. Summary of Risk Assessments](#bookmark22) 16

[6. Create Policies for Encryption and Triple A](#bookmark23) 16

[7. Map the Principles](#bookmark24) 17

[Audit Controls and Management](#bookmark25) 18

[Enforcement](#bookmark26) 18

[Exceptions Process](#bookmark27) 18

[Distribution](#bookmark28) 19

[Policy Change Control](#bookmark29) 19

[Policy Version History](#bookmark30) 19

[Appendix A Lookups](#bookmark31) 19

[Approved C/C++ Language Acronyms](#bookmark32) 19

## 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 | Verify and sanitize input to prevent vulnerabilities such as injection attacks or data corruption. |
| 1. Heed Compiler Warnings | Pay attention to compiler warnings to catch potential security issues and ensure code integrity. |
| 1. Architect and Design for Security Policies | Integrate security into the software architecture and design to enforce policy compliance and protect against threats. |
| 1. Keep It Simple | Minimize complexity to reduce vulnerabilities and make security measures easier to implement and maintain. |
| 1. Default Deny | Only allow access explicitly and deny by default, reducing the risk of unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Provide minimal privileges required for tasks, limiting potential damage in the event of a security breach. |
| 1. Sanitize Data Sent to Other Systems | Clean and validate data before sending to prevent security issues such as data corruption or system compromise. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls to create a robust and comprehensive defense strategy against various threats. |
| 1. Use Effective Quality Assurance Techniques | Use robust testing and code review processes for security to ensure the reliability and integrity of the software. |
| 1. Adopt a Secure Coding Standard | Follow established coding standards for secure development, providing guidelines and best practices to minimize vulnerabilities and promote secure coding habits. |

### 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** | **Data Type Consistency** |
| --- | --- | --- |
| **Data Type** | STD-001-DTP | Ensure proper use of data types to improve code reliability, prevent type-related vulnerabilities, and ensure data integrity and correctness. |

| **Noncompliant Code** |
| --- |
| The use of the **int** data type for a large number can lead to potential overflow or truncation issues. |
| int i = 10; // Noncompliant: Inadequate data type for a large number |

| **Compliant Code** |
| --- |
| The use of the **long long int** data type ensures sufficient range for storing the large number without the risk of overflow or truncation. |
| long long int i = 10; // Compliant: Use a larger data type to accommodate the large number |

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

|  |
| --- |
| **Principles(s):** Maintainability - By adhering to proper data type selection, we ensure the code's maintainability by reducing potential issues related to data integrity and type-related vulnerabilities. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeAnalyzer | Version: 2.5.1 | DataTypeCheck | CodeAnalyzer's DataTypeCheck enforces proper data type usage, reducing the risk of vulnerabilities and enhancing code reliability. |
| LintMaster | *Version:* 3.0.2 | TypeSafetyGuard | LintMaster's TypeSafetyGuard detects and reports potential type-related vulnerabilities, helping ensure proper data type usage and enhancing code reliability. |
| CodeGuardian | Version: 1.8.0 | TypeIntegrityCheck | CodeGuardian's TypeIntegrityCheck identifies and prevents type-related issues, improving data integrity and correctness by enforcing consistent data type usage. |
| SecureCodeAnalyzer | Version: 4.2.3 | DataTypeGuard | SecureCodeAnalyzer's DataTypeGuard scans code for improper data type usage, reducing the risk of vulnerabilities and enhancing code reliability. It enforces consistent data type selection to prevent type-related issues. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Validation** |
| --- | --- | --- |
| **Data Value** | STD-002-DVL | Validate and sanitize data values to ensure integrity, prevent invalid or malicious inputs, and mitigate potential vulnerabilities arising from untrusted or unexpected data. |

| **Noncompliant Code** |
| --- |
| The use of a negative value for the age data violates the expected range and can lead to incorrect program behavior. |
| int age = -10; // Noncompliant: Invalid data value for age |

| **Compliant Code** |
| --- |
| The use of an unsigned data type for age ensures that only positive values within the expected range are accepted. |
| unsigned int age = 30; // Compliant: Use an unsigned data type to ensure positive age values |

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

|  |
| --- |
| **Principles(s):** Security - By validating and sanitizing data values, we enhance the security of our code by preventing invalid or malicious inputs from compromising the integrity and behavior of the program. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Low | Medium | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| DataGuard | Version: 2.1.0 | DataValueValidation | DataGuard's DataValueValidation enforces proper validation of data values, reducing the risk of vulnerabilities and enhancing data integrity. |
| InputScanner | Version: 3.5.2 | MaliciousInputCheck | InputScanner's MaliciousInputCheck detects and prevents malicious data inputs, ensuring that only valid and safe data values are accepted. |
| SafeCoders | Version: 1.0.7 | ValueSanitizationGuard | SafeCoders' ValueSanitizationGuard identifies and sanitizes data values, mitigating potential vulnerabilities arising from untrusted or unexpected data. |
| IntegrityShield | Version: 4.0.1 | RangeValidation | IntegrityShield's RangeValidation ensures that data values fall within expected ranges, preventing issues caused by out-of-bound values and enhancing data integrity. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness** |
| --- | --- | --- |
| **String Correctness** | STD-003-SCR | Properly handle and validate strings to ensure correct behavior, prevent buffer overflows, format string vulnerabilities, and mitigate security risks associated with improper string manipulation. |

| **Noncompliant Code** |
| --- |
| The use of **strcpy** to copy a long string into a small buffer can lead to buffer overflow vulnerabilities. |
| char buffer[10];  strcpy(buffer, "This is a very long string that exceeds the buffer size"); // Noncompliant: Buffer overflow risk |

| **Compliant Code** |
| --- |
| The use of **strncpy** with the specified buffer size ensures that the string is properly truncated to fit within the buffer, preventing buffer overflows. |
| char buffer[20];  strncpy(buffer, "This is a shorter string", sizeof(buffer) - 1); // Compliant: Use strncpy and specify buffer size to prevent overflow  buffer[sizeof(buffer) - 1] = '\0'; // Null-terminate the string |

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

|  |
| --- |
| **Principles(s):** Reliability - By properly handling and validating strings, we enhance the reliability of our code by preventing buffer overflows and other vulnerabilities that can lead to incorrect behavior or security risks. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SecureStrings | Version: 3.2.1 | StringValidationGuard | SecureStrings' StringValidationGuard enforces proper handling and validation of strings, reducing the risk of buffer overflows and format string vulnerabilities. |
| StringDefender | Version: 2.0.5 | BufferSizeCheck | StringDefender's BufferSizeCheck ensures that string functions are used with correct buffer sizes, preventing buffer overflows and enhancing string correctness. |
| StrSafe | Version: 1.8.3 | StringManipulationGuard | StrSafe's StringManipulationGuard detects improper string manipulation, enhancing code reliability and mitigating security risks associated with strings. |
| SafeStr | Version: 4.5.0 | FormatStringProtection | SafeStr's FormatStringProtection guards against format string vulnerabilities, ensuring proper string handling and mitigating security risks. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Prevention** |
| --- | --- | --- |
| **SQL Injection** | STD-004-SQL | Implement measures to prevent SQL injection attacks by properly sanitizing user input, using parameterized queries or prepared statements, and avoiding dynamic SQL construction. |

| **Noncompliant Code** |
| --- |
| The code constructs SQL queries by directly concatenating user input, making it vulnerable to SQL injection attacks. |
| string username = "admin'; DROP TABLE Users; --";  string query = "SELECT \* FROM Users WHERE username = '" + username + "';";  executeQuery(query); // Noncompliant: Vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| The code uses prepared statements with parameter binding to separate user input from the SQL query, preventing SQL injection vulnerabilities. |
| string username = "admin";  string query = "SELECT \* FROM Users WHERE username = ?;";  prepareStatement(query);  executePreparedStatement(query, username); // Compliant: Use prepared statements to avoid SQL injection |

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

|  |
| --- |
| **Principles(s):** Security - By implementing proper SQL injection prevention measures, we enhance the security of our code by mitigating the risk of unauthorized data access and manipulation. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SecureSQL | Version: 2.1.0 | SQLInjectionGuard | SecureSQL's SQLInjectionGuard detects and prevents SQL injection vulnerabilities by promoting the use of parameterized queries and prepared statements. |
| SQLShield | Version: 3.5.2 | InputSanitizationCheck | SQLShield's InputSanitizationCheck ensures proper sanitization of user input, preventing SQL injection vulnerabilities and enhancing database security. |
| SQLGuardian | Version: 1.0.7 | DynamicSQLPrevention | SQLGuardian's DynamicSQLPrevention identifies and prevents dynamic SQL construction, reducing the risk of SQL injection attacks. |
| InjectionDefender | Version: 4.0.1 | QueryParameterization | InjectionDefender's QueryParameterization enforces the use of parameterized queries, mitigating SQL injection risks and enhancing code security. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection** |
| --- | --- | --- |
| **Memory Protection** | STD-005-MEM | Implement measures to ensure memory safety, prevent buffer overflows, memory leaks, and other memory-related vulnerabilities, and promote secure memory management practices. |

| **Noncompliant Code** |
| --- |
| The use of unsafe functions like **gets** can lead to buffer overflow vulnerabilities and compromise memory integrity. |
| char buffer[10];  gets(buffer); // Noncompliant: Potential buffer overflow |

| **Compliant Code** |
| --- |
| The use of **fgets** with the specified buffer size ensures that input is limited to the available buffer space, preventing buffer overflows and maintaining memory safety. |
| char buffer[10];  fgets(buffer, sizeof(buffer), stdin); // Compliant: Use safer functions like fgets with size limit |

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

|  |
| --- |
| **Principles(s):** Reliability - By implementing memory protection measures, we enhance the reliability of our code by preventing memory-related vulnerabilities that can lead to crashes, data corruption, and security risks. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SecureMem | Version: 2.1.0 | BufferSafetyGuard | SecureMem's BufferSafetyGuard enforces secure memory management practices, reducing the risk of buffer overflows and promoting memory safety. |
| MemShield | Version: 3.5.2 | LeakDetection | MemShield's LeakDetection identifies memory leaks, ensuring proper memory deallocation and preventing resource wastage. |
| SafeAllocator | Version: 1.0.7 | MemoryAllocationCheck | SafeAllocator's MemoryAllocationCheck enforces proper memory allocation and deallocation, mitigating memory-related vulnerabilities. |
| MemoryGuardian | Version: 4.0.1 | BoundsChecking | MemoryGuardian's BoundsChecking detects and prevents buffer overflows, maintaining memory integrity and enhancing code reliability. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions Usage** |
| --- | --- | --- |
| **Assertions** | STD-006-AST | Use assertions to validate assumptions, verify invariants, and detect programming errors during development and testing, aiding in identifying and resolving issues early. |

| **Noncompliant Code** |
| --- |
| The assertion fails due to an incorrect assumption, indicating a programming error. |
| int x = 10;  assert(x > 20); // Noncompliant: Assertion failure, incorrect assumption |

| **Compliant Code** |
| --- |
| The assertion verifies that the value of **x** is non-negative, ensuring the expected invariant is maintained. |
| int x = 10;  assert(x >= 0); // Compliant: Assertion validates an expected invariant |

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

|  |
| --- |
| **Principles(s):** [Name the principle and explain how it maps to this standard.] |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Low | Medium | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AssertGuard | Version: 2.1.0 | AssertionValidation | AssertGuard's AssertionValidation promotes the use of assertions for assumptions and invariants, aiding in identifying programming errors during development and testing. |
| InvariantVerifier | Version: 3.5.2 | InvariantCheck | InvariantVerifier's InvariantCheck validates invariants using assertions, enhancing code reliability by detecting violations early in the development process. |
| ErrorDetector | Version: 1.0.7 | AssertionErrorCheck | ErrorDetector's AssertionErrorCheck identifies assertion failures, helping developers detect and resolve programming errors early in the development cycle. |
| AssertInspector | Version: 4.0.1 | AssertionCoverage | AssertInspector's AssertionCoverage measures assertion coverage, providing insights into the effectiveness of assertions in identifying and resolving issues. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exception Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-EXC | Use appropriate exception handling mechanisms to handle exceptional conditions, promote error recovery, and ensure proper program behavior in the presence of exceptions. |

| **Noncompliant Code** |
| --- |
| The catch-all exception handler (**catch (...)**) can lead to obscured errors and hinder proper error handling and debugging. |
| try {  // Code that may throw an exception  // ...  }  catch (...) {  // Noncompliant: Catch-all exception handler  // ...  } |

| **Compliant Code** |
| --- |
| The code catches a specific exception type (**SpecificException**) to handle and respond to the expected exceptional condition appropriately. |
| try {  // Code that may throw an exception  // ...  }  catch (const SpecificException& ex) {  // Compliant: Catch specific exception type  // ...  } |

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

|  |
| --- |
| **Principles(s):** Robustness - By using appropriate exception handling mechanisms, we enhance the robustness of our code by ensuring that exceptional conditions are handled properly, promoting error recovery, and improving program behavior in the presence of exceptions. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ExceptionGuard | Version: 2.1.0 | ExceptionTypeCheck | ExceptionGuard's ExceptionTypeCheck enforces the use of specific exception types, reducing catch-all exception handlers and improving error handling. |
| ExceptionInspector | Version: 3.5.2 | ExceptionCoverage | ExceptionInspector's ExceptionCoverage measures exception coverage, providing insights into the effectiveness of exception handling mechanisms. |
| ErrorRecoveryHelper | Version: 1.0.7 | ErrorRecoveryCheck | ErrorRecoveryHelper's ErrorRecoveryCheck ensures that exception handling promotes proper error recovery and program behavior in exceptional conditions. |
| ExceptionDebugger | Version: 4.0.1 | ExceptionDebugging | ExceptionDebugger's ExceptionDebugging assists in debugging exception-related issues, enhancing the identification and resolution of exception handling problems. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Input Validation** |
| --- | --- | --- |
| **INPUT** | STD-008-IVL | Validate and sanitize user input to ensure it meets expected criteria, prevent malicious inputs, and mitigate security vulnerabilities such as injection attacks or data corruption. |

| **Noncompliant Code** |
| --- |
| The use of a negative value for quantity violates the expected range and can lead to incorrect program behavior or vulnerabilities. |
| int quantity = -10; // Noncompliant: Invalid input value |

| **Compliant Code** |
| --- |
| The code retrieves a validated quantity value using a custom function that ensures the input meets the required criteria. |
| int quantity = getValidatedQuantity(); // Compliant: Input is validated against expected criteria |

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

|  |
| --- |
| **Principles(s):** Security - By validating and sanitizing user input, we enhance the security of our code by preventing malicious inputs and reducing the risk of security vulnerabilities such as injection attacks or data corruption. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SecureInput | Version: 2.1.0 | InputValidationGuard | SecureInput's InputValidationGuard enforces proper validation and sanitization of user input, reducing the risk of security vulnerabilities. |
| InputDefender | Version: 3.5.2 | InputSanitizationCheck | InputDefender's InputSanitizationCheck detects and prevents improper input, ensuring that user-provided data adheres to expected criteria. |
| SafeInputChecker | Version: 1.0.7 | InputRangeCheck | SafeInputChecker's InputRangeCheck enforces input range validation, preventing issues caused by out-of-bound values and enhancing data integrity. |
| InputGuardian | Version: 4.0.1 | MaliciousInputDetection | InputGuardian's MaliciousInputDetection identifies and prevents malicious input, mitigating security risks and promoting secure user input handling. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Secure Random Number Generation** |
| --- | --- | --- |
| **Random Number** | STD-009-RNG | [Rationalize the standard.] |

| **Noncompliant Code** |
| --- |
| The use of **rand()** for generating random values is not suitable for security-sensitive operations as it lacks cryptographic strength. |
| int randomValue = rand(); // Noncompliant: Insecure random number generation |

| **Compliant Code** |
| --- |
| The code utilizes a secure random number generation function that ensures the generated values possess cryptographic strength. |
| int randomValue = generateSecureRandomValue(); // Compliant: Use a secure random number generation method |

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

|  |
| --- |
| **Principles(s):** Security - By utilizing secure random number generation methods, we enhance the security of our code by ensuring that generated values possess cryptographic strength, suitable for security-sensitive operations. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SecureRNG | Version: 2.1.0 | SecureRandomCheck | SecureRNG's SecureRandomCheck enforces the use of secure random number generation methods, reducing the risk of using weak random values in security-sensitive operations. |
| CryptoNumberGen | Version: 3.5.2 | CryptographicStrengthCheck | CryptoNumberGen's CryptographicStrengthCheck verifies the cryptographic strength of random number generation methods, promoting secure random value usage. |
| RandomAudit | Version: 4.0.1 | RandomValueAudit | RandomAudit's RandomValueAudit assesses the quality of random number generation, providing insights into the suitability of generated values for security-sensitive operations. |
| SecureCodeGuardian | Version: 1.0.7 | RandomnessAssessment | SecureCodeGuardian's RandomnessAssessment evaluates the randomness of generated values, ensuring cryptographic strength and mitigating security risks. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Error Handling and Logging** |
| --- | --- | --- |
| **ERROR HANDLING** | STD-010-EHL | Implement appropriate error handling mechanisms and logging practices to capture and handle errors, aid in debugging, and provide necessary information for troubleshooting and security analysis. |

| **Noncompliant Code** |
| --- |
| The code lacks error handling and logging, making it difficult to identify or troubleshoot errors that may occur during the operation. |
| // Noncompliant: No error handling or logging  int result = performOperation(); |

| **Compliant Code** |
| --- |
| The code uses a try-catch block to handle exceptions and logs any encountered errors, aiding in error identification and providing relevant information for troubleshooting and analysis. |
| // Compliant: Error handling and logging  try {  int result = performOperation();  }  catch (const Exception& ex) {  logError(ex);  // Handle or propagate the error as appropriate  } |

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

|  |
| --- |
| **Principles(s):** Reliability and Debugging - By implementing effective error handling and logging practices, we enhance the reliability of our code by capturing and handling errors, aiding in debugging, and facilitating troubleshooting and security analysis. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ErrorLogger | Version: 2.1.0 | ErrorHandlingCheck | ErrorLogger's ErrorHandlingCheck enforces the use of appropriate error handling mechanisms and logging practices, promoting code reliability and aiding in debugging. |
| LogGuardian | Version: 3.5.2 | LoggingBestPractices | LogGuardian's LoggingBestPractices checks logging practices to ensure relevant error information is captured, aiding in troubleshooting and security analysis. |
| DebugHelper | Version: 1.0.7 | ErrorDebuggingCheck | DebugHelper's ErrorDebuggingCheck assists in error debugging, providing insights into error handling and facilitating error resolution. |
| ErrorAnalyzer | 4.0.1 | ErrorAnalysis | ErrorAnalyzer's ErrorAnalysis assesses error handling practices, identifying potential issues and suggesting improvements for better error capture and resolution. |

### 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.

Pre-production: Assess, Design, Build, Verify

In pre-production, integrate automated coding standards checks into our CI/CD pipeline. Use static analysis tools during design and build phases to catch violations early. Verify compliance during testing using dynamic analysis and security tools.

Production: Transition, Monitor, Respond, Maintain

During production, automate checks for coding standards alignment in the live environment. Monitor code continuously for compliance and security issues. Set up alerts to respond to violations. Maintain code quality with regular automated reviews.

By automating coding standards enforcement throughout the DevSecOps process, we ensure our code meets standards, enhances security, and minimizes vulnerabilities.

### 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-001-DTP | High | Likely | Medium | High | 4 |
| STD-002-DVL | Medium | Possible | Low | Medium | 2 |
| STD-003-SCR | High | Likely | Medium | High | 4 |
| STD-004-SQL | High | Likely | Medium | High | 5 |
| STD-005-MEM | High | Likely | Medium | High | 5 |
| STD-006-AST | Medium | Possible | Low | Medium | 2 |
| STD-007-EXC | High | Likely | Medium | High | 4 |
| STD-008-IVL | High | Likely | Medium | High | 3 |
| STD-009-RNG | High | Likely | Medium | High | 5 |
| STD-010-EHL | High | Likely | Medium | High | 4 |

### 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 in rest | At-rest encryption safeguards data when it is stored in databases, servers, or storage devices. It prevents unauthorized access to data even if the physical storage medium is compromised. |
| Encryption at flight | In-flight encryption secures data during transmission over networks or communication channels. It ensures that data is protected against interception and unauthorized access while being transferred between systems or devices. |
| Encryption in use | In-use encryption protects data during processing and manipulation, ensuring that sensitive information remains encrypted even during computations or operations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users before granting access to systems or data. It ensures that only authorized individuals can gain entry. |
| Authorization | Authorization defines the level of access and permissions granted to authenticated users. It ensures that users have appropriate rights based on their roles. |
| Accounting | Accounting involves tracking and logging user activities, providing an audit trail of who accessed what data and when. It enhances transparency 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 a best practice.

Coding Standard 1: Data Type Consistency

Principle(s): 1, 5

* **Robustness:** Making sure we use the right data types improves how dependable our code is, stopping type-related problems and security holes. This also keeps our data safe from getting messed up.
* **Simplicity:** When we use the correct data types, our code is easier to manage, which means fewer mistakes and a more secure program overall.

Coding Standard 2: Data Value Validation

Principle(s): 1, 3

* **Robustness:** Checking and cleaning up data values keeps us safe from invalid or bad inputs, so our code isn't vulnerable to sneaky attacks or messed-up data.
* **Least Privilege:** Validating data in the right way supports the idea of giving only the necessary permissions to users, which makes things more secure by limiting what can go wrong.

Coding Standard 3: String Correctness

Principle(s): 1, 6

* **Robustness:** Handling strings properly and checking them makes sure we don't get into trouble with bugs or attacks that target our strings, keeping our code solid and secure.
* **Defense in Depth:** By nailing down how we manage strings, we're building up layers of security to protect against potential problems and attacks on our code.

Coding Standard 4: SQL Injection Prevention

Principle(s): 1, 2

* **Robustness:** Stopping SQL injection attacks boosts how reliable and secure our code is by closing a common way hackers try to sneak in.
* **Fail-Safe Defaults:** Using parameterized queries and prepared statements means we're playing it safe from the start, which makes sure our data stays safe and private.

Coding Standard 5: Memory Protection

Principle(s): 1, 6

* **Robustness:** Safeguarding against memory problems and vulnerabilities keeps our code strong and prevents any risky business from happening due to memory issues.
* **Defense in Depth:** Looking after our code's memory protection is like putting on layers of armor – it shields against different kinds of attacks that target the memory.

Coding Standard 6: Assertions Usage

Principle(s): 1, 4

* **Robustness:** Using checks to catch mistakes and problems boosts the reliability of our code, helping us spot issues early.
* **Fail-Safe Defaults:** Assertions act like early-warning systems, catching things that might cause problems or create security holes, making our code more reliable from the get-go.

Coding Standard 7: Exception Handling

Principle(s): 1, 4

* **Robustness:** Setting up the right way to deal with exceptions helps keep our code reliable, even when things don't go as planned.
* **Fail-Safe Defaults:** Handling exceptions properly ensures we're not leaving our code exposed to unexpected behavior that could lead to security problems.

Coding Standard 8: Input Validation

Principle(s): 1, 2

* **Robustness:** Checking and cleaning up what users put into our systems boosts how solid our code is and makes sure we're not vulnerable to bad stuff getting in.
* **Fail-Safe Defaults:** By filtering out potentially harmful inputs, we're creating a solid foundation that keeps our code safe from the start and minimizes potential attacks.

Coding Standard 9: Secure Random Number Generation

Principle(s): 1, 3

* **Robustness:** Using strong methods to create random numbers makes sure our code is dependable, especially for things that need to be super secure.
* **Least Privilege:** Secure random number generation ensures only the right processes can use these super important random numbers, keeping them safe from being misused.

Coding Standard 10: Error Handling and Logging

Principle(s): 1, 6

* **Robustness:** Setting up ways to catch and log errors helps keep our code solid by spotting problems and making it easier to figure out what went wrong.
* **Defense in Depth:** Error handling and logging are like having extra security layers – they help us find out what's happening and who's responsible, so we can respond and protect our code better.

By linking the principles to each coding standard, we're showing how we've built our security policy on solid principles, making sure our coding standards improve our code's reliability and security.

**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 |  |
| **[Insert text.]** | 08/02/2023 | Update Template Project 1 | Roderick Fisher | [Insert text.] |
| **[Insert text.]** | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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