**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 | Validating input from untrusted data sources, especially external data sources. This will block any malicious injection within an application which will decrease the number of vulnerabilities. |
| 1. Heed Compiler Warnings | Compiling at the highest level can catch any additional vulnerabilities. A mix of static and dynamic testing tools can aid while compiling code to catch additional vulnerabilities. |
| 1. Architect and Design for Security Policies | The architecture of the software must be thoughtfully planned out. During implementation, taking into account the different security methods that will be utilized should be taken into account so the code is organized and can be easily maintained and modified. |
| 1. Keep It Simple | Keeping the application simply allows for easier to maintain and also decreases the number of possible errors. Ensuring that there are only necessary layers of security also decreases operational costs. |
| 1. Default Deny | By default, permission should be denied in an application, the application will then gain access based on the permission. |
| 1. Adhere to the Principle of Least Privilege | Each application should require the least amount of permission to execute the program. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems ensures that hackers cannot use unused functionality to inject malicious SQL. |
| 1. Practice Defense in Depth | Depth ensures that if one security measure fails another is able to catch and prevent or mitigate the vulnerability/attack. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance can identify and mitigate possible vulnerabilities with techniques such as vulnerability scans and static testing. |
| 1. Adopt a Secure Coding Standard | Developing and applying secure coding standards adds to the security and maintainability of code, which is essential for identifying and mitigating security 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] | Use of correct data types to prevent overflow or underflow. |

| **Noncompliant Code** |
| --- |
| This block of code uses char data type instead of other more appropriate data types for arithmetic. This change will cause an int overflow, which can create potential security vulnerabilities. |
| char val1 = 800;  char val2 = 600;  char value = val1 + val2;  cout<< “Total:” << (int)result <<endl; //Outputs 120 due to overflow |

| **Compliant Code** |
| --- |
| Instead this code block uses the int data type which can the values and therefore the arithmetic. This will prevent overflow and display the correct value of 1400, instead of 120. |
| int val1 = 800;  int val2 = 600;  int value = val1 + val2;  cout << "Total:" << (int)value << endl; // This will output 1400 |

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

| **Principles(s):** INT50-CPP: prevents casting an out-of-range value which prevents crashes. Using the correct types prevents overflow, underflow, and any other unexpected behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Probable(2) | Low(3) | Medium P18 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S5945  <https://rules.sonarsource.com/cpp/RSPEC-5945/> | This tool detects unsafe type conversions |
| SonarQube | 10.3+ | cpp:S5276  <https://rules.sonarsource.com/cpp/RSPEC-5276/> | This tool identifies incorrect assumptions of types. |
| SonarQube | 10.3+ | cpp:S3584  <https://rules.sonarsource.com/cpp/RSPEC-3584/> | This tool identifies data conversions that can cause data loss. |
| SonarQube | 10.3+ | java:S2184  <https://rules.sonarsource.com/java/RSPEC-2184/> | This tool detects integer division precision loss. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating input to prevent malicious input/injection. |

| **Noncompliant Code** |
| --- |
| This code block uses the input without proper validation, which can cause vulnerabilities or overflow which can be malicious to the application. |
| char buffer[50];  string userInput;  cout << "Enter your cat’s name: ";  getline(cin, userInput);  strcpy(buffer, userInput.c\_str()); // No proper validation  cout << "Cat’s name: " << buffer << endl; |

| **Compliant Code** |
| --- |
| This code block demonstrates proper input validation, ensuring that the input integrity and also prevents other vulnerabilities. This validation ensures that the input is less than 50 characters and constraints the input to only using letters and no special characters that are commonly used for injection. |
| char buffer[50];  string userInput;  cout << "Enter your cat’s name: ";  getline(cin, userInput);  if (userInput.length() < 50 && userInput.find\_first\_not\_of("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ") == string::npos) {  strcpy(buffer, userInput.c\_str()); // Validation  cout << "Cat’s name: " << buffer << endl;  } else {  cout << "Invalid input" << endl;  } |

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

| **Principles(s):** IDS50-CPP Do not use vulnerable input validation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Likely(3) | Medium(3) | High P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S2183  <https://rules.sonarsource.com/cpp/RSPEC-2183> | This tool helps detect integer overflow in operations. |
| SonarQube | 10.3+ | cpp:S2184  <https://rules.sonarsource.com/cpp/RSPEC-2184> | This tool helps identify operations that may cause overflow. |
| SonarQube | 10.3+ | cpp:S3519  <https://rules.sonarsource.com/cpp/RSPEC-3519> | This tool identifies unvalidated data values from external sources. |
| SonarQube | 10.3+ | java:S2676  <https://rules.sonarsource.com/java/RSPEC-2676> | This tool helps detect infinity on float values. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CCP] | Ensures that the strings are properly handled to prevent corruption or buffer overflow. |

| **Noncompliant Code** |
| --- |
| This code block uses unsafe string functions that do not check buffer boundaries which can cause buffer overflows along with other vulnerabilities. In this example the destination can accept 10 characters however the source is larger than 10 characters which would then cause a buffer overflow. |
| char destination[10];  char source[] = "Thghshxbdnbvhjdbhdvjfbvjncjkvbddfhjdfbdv";  strcpy(destination, source); // Buffer overflow exceeds destination  strcat(destination, " more text");  cout << destination << endl; |

| **Compliant Code** |
| --- |
| The code block implements a string function which checks the boundaries to prevent overflows and other vulnerabilities. |
| char destination[50];  char source[] = "gfshvshvsdhvhd";  strncpy(destination, source, sizeof(destination) - 1); // Copy with size limit  destination[sizeof(destination) - 1] = '\0'; // Ensure null termination  strncat(destination, " more text", sizeof(destination) - strlen(destination) - 1); // Safe concatenation  cout << destination << endl; |

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

| **Principles(s): INT32-C: Ensures that operations do not result in overflow. This helps validate operations before an operation runs to ensure that no undefined behavior occurs.** |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Probable(2) | Low(3) | Medium P18 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S5547  <https://rules.sonarsource.com/cpp/RSPEC-5547> | This tool detects unsafe string functions. |
| SonarQube | 10.3+ | cpp:S3584  <https://rules.sonarsource.com/cpp/RSPEC-3584> | This tool helps identify buffer overflow in string operations. |
| SonarQube | 10.3+ | cpp:S6069  <https://rules.sonarsource.com/cpp/RSPEC-6069> | This tool helps string comparisons with which methods should be used. |
| SonarQube | 10.3+ | java:S1149  <https://rules.sonarsource.com/java/RSPEC-1149> | This tool detects synchronized class usage in strings. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Parametrizes queries which help prevent  SQL injections. |

| **Noncompliant Code** |
| --- |
| This block of code demonstrates non-compliant practices since it concatenates the user input, which can be malicious, into the SQl query. This would make the code susceptible to SQL injections. |
| string username, password;  cout << "Enter username: ";  cin >> username;  cout << "Enter password: ";  cin >> password;  string query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'";  // the user could inject a malicious input directly into the query |

| **Compliant Code** |
| --- |
| This code block shows the use of parameterized queries, in this case the use of placeholders, which prevent malicious SQL injection. |
| string username, password;  cout << "Enter username: ";  cin >> username;  cout << "Enter password: ";  cin >> password;  // Parameterized queries  string query = "SELECT \* FROM users WHERE username = ? AND password = ?";  // Execute with bound parameters prevents SQL injection |

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

| **Principles(s):** IDS01-J: Ensures that the strings are normalized before validation. This principle helps prevent SQL injections since it ensures that the input is sanitized. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Likely(3) | Medium(3) | Low P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S2077  <https://rules.sonarsource.com/cpp/RSPEC-2077> | This tool helps detect SQL injection vulnerabilities from strings. |
| SonarQube | 10.3+ | cpp:S3649 <https://rules.sonarsource.com/cpp/RSPEC-3649> | This tool detects database queries which are susceptible to SQL attacks. |
| SonarQube | 10.3+ | java:S2077  <https://rules.sonarsource.com/java/RSPEC-2077> | This tool helps identify SQL injection risks in queries. |
| SonarQube | 10.3+ | java:S3649  <https://rules.sonarsource.com/java/RSPEC-3649> | This tool helps identify LDAP injection vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Managing memory allocation/deallocation to prevent memory corruption and leaks. |

| **Noncompliant Code** |
| --- |
| This block of code depicts memory allocation; however, it does not deallocate, which can give access to the free memory and even cause potential leaks. |
| int\* data = new int[100];  // Uses the memory  for(int i = 0; i < 100; i++) {  data[i] = i;  }  // Memory is not deallocated |

| **Compliant Code** |
| --- |
| This block of code corrects the deallocation. Not only does the code allocate memory but ensures that for every allocation there is a deallocation of memory to prevent such vulnerabilities. |
| int\* data = new int[100];  // Uses the memory  for(int i = 0; i < 100; i++) {  data[i] = i;  }  delete[] data; // Properly frees the allocated memory(deallocation)  data = nullptr; // Prevents re use of memory |

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

| **Principles(s):** MEM31-C: This principle frees dynamic memory when it is no longer in use. This ensures that memory is managed correctly, which helps prevent memory leaks or exploitations by pointers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium(2) | Probable(2) | Low(3) | Medium P12 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S3584  <https://rules.sonarsource.com/cpp/RSPEC-3584> | This tool helps detect unsafe memory operations. |
| SonarQube | 10.3+ | cpp:S5827  <https://rules.sonarsource.com/cpp/RSPEC-5827> | This tool helps identify vulnerabilities such as use after free memory. |
| SonarQube | 10.3+ | cpp:S3584  <https://rules.sonarsource.com/cpp/RSPEC-3584> | This tool helps detect memory leaks from resource management. |
| SonarQube | 10.3+ | java:S2095  <https://rules.sonarsource.com/java/RSPEC-2095> | This tool identifies resources that should be closed. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | The use of assertions will help validate assumptions and also errors during the development process. |

| **Noncompliant Code** |
| --- |
| The code block fails to validate assumptions, passing on invalid conditions which can cause security vulnerabilities or crashes. |
| #include <iostream>  using namespace std;  int divide(int a, int b) {  // No validation of assumptions  return a / b; // Division by zero will cause crash  }  int main() {  int result = divide(10, 0); // Undefined behavior  cout << "Result: " << result << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This code block does use assertions to validate assumptions and catches errors early on. |
| #include <iostream>  #include <cassert>  using namespace std;  int divide(int a, int b) {  assert(b != 0); // Validates assumption that divisor is not 0  return a / b; // Safe division after validation  }  int main() {  int result = divide(10, 5); // Valid operation, wont crash  cout << "Result: " << result << endl;  return 0;  } |

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

| **Principles(s):** MSC11-C: This principle incorporates diagnostic testing with the use of assertions. This helps validate assumptions which can catch errors before execution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Likely(3) | High(1) | High P9 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S3584  <https://rules.sonarsource.com/cpp/RSPEC-3584> | This tool helps detect missing precondition checks. |
| SonarQube | 10.3+ | cpp:S1145  <https://rules.sonarsource.com/cpp/RSPEC-1145> | This tool is useful for identifying if (true/false) statements. |
| SonarQube | 10.3+ | cpp:S5443  <https://rules.sonarsource.com/cpp/RSPEC-5443> | This tool identifies operations that should be validated using assertions. |
| SonarQube | 10.3+ | java:S3415  <https://rules.sonarsource.com/java/RSPEC-3415> | This tool helps detect arguments that are not validated. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception handling will manage any errors gracefully without crashing the program. |

| **Noncompliant Code** |
| --- |
| This code block does not candle possible exceptions, which will make the program crash and can lead to sensitive information exposure. |
| int main() {  vector<int> numbers = {1, 2, 3};  // No exception handling, will cause crash  cout << numbers.at(10) << endl; // Out of range access crashes program  return 0;  } |

| **Compliant Code** |
| --- |
| This code block handles exceptions gracefully by the use of try-catch blocks. This will prevent errors, even those that may crash the application. |
| int main() {  vector<int> numbers = {1, 2, 3};  try {  cout << numbers.at(10) << endl; // Potential exception  }  catch(const out\_of\_range& e) {  cout << "Error: Index out of range" << endl; // error handling  }  return 0;  } |

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

| **Principles(s):** ERR00-J This principle does not ignore checked exceptions, which enforces the use of exception handling. This helps prevent unexpected termination, or improper exposure of system information. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Probable(2) | Medium (3) | Medium P18 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S1005  <https://rules.sonarsource.com/cpp/RSPEC-1005> | This tool helps detect empty catch blocks. |
| SonarQube | 10.3+ | cpp:S5945  <https://rules.sonarsource.com/cpp/RSPEC-5945> | This tool helps detect exceptions thrown in destructors. |
| SonarQube | 10.3+ | java:S1181  <https://rules.sonarsource.com/java/RSPEC-1181> | This tool helps identify broad exceptions. |
| SonarQube | 10.3+ | java:S1130  <https://rules.sonarsource.com/java/RSPEC-1130> | This tool helps identify unnecessary exception declarations. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure File Handling | [STD-008-CPP] | Standard validates the path and implements secure file management to ensure there is no unauthorized access. |

| **Noncompliant Code** |
| --- |
| This code takes the users input directly meaning it is not validated, which makes the application susceptible to path traversal attacks and unauthorized access. |
| string filename;  cout << "Enter filename to read: ";  cin >> filename;  // No path validation making it vulnerable to path traversal  ifstream file(filename); // User could input malicious path  if (file.is\_open()) {  string line;  while (getline(file, line)) {  cout << line << endl; // Could expose files  }  } |

| **Compliant Code** |
| --- |
| This code block validates the file path, ensuring that no traversal occurs and limits files access to safe directories. |
| string filename;  cout << "Enter filename to read: ";  cin >> filename;  // Validate filename  if (filename.find("..") != string::npos || filename.find("/") != string::npos) {  cout << "Invalid filename" << endl;  return -1;  }  string safePath = "./safe\_directory/" + filename; // Restrict to safe directory  ifstream file(safePath);  if (file.is\_open()) {  string line;  while (getline(file, line)) {  cout << line << endl;  }  } |

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

| **Principles(s):** FIO16-J: This principle canonicalizes path names before validation. This helps ensure that paths are validated and helps prevent unauthorized file access and helps sensitive information from being exposed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Likely(3) | Medium(2) | High P18 | L1 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| SonarQube | 10.3+ | cpp:S134  <https://rules.sonarsource.com/cpp/RSPEC-134> | This tool helps detect control flow statements that are nested deep. |
| SonarQube | 10.3+ | cpp:S1541  <https://rules.sonarsource.com/cpp/RSPEC-1541> | This tool helps identify functions with high complexity. |
| SonarQube | 10.3+ | cpp:S3776  <https://rules.sonarsource.com/cpp/RSPEC-3776> | This tool helps identify functions with cyclomatic complexity. |
| SonarQube | 10.3+ | java:S1151  <https://rules.sonarsource.com/java/RSPEC-1151> | This tool helps detect switch statements with too many cases. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Random Number Generator | [STD-009-CPP] | Use of cryptographically secure random number generators for secure operations. |

| **Noncompliant Code** |
| --- |
| This code uses predictable pseudo-random number generation that can be easily guessed by attackers, making it risky in regard to security. |
| srand(time(0)); // Predictable seed  int sessionToken = rand() % 1000000; // Weak random generation, easily predicted  cout << "Session token: " << sessionToken << endl;  // Attackers can predict this token easily, not secure enough |

| **Compliant Code** |
| --- |
| This code uses cryptographically secure random number generation that produces unpredictable values making it more secure and less guessable. |
| random\_device rd;  mt19937 gen(rd()); // Seed with secure random generator  uniform\_int\_distribution<> dis(100000, 999999);  int sessionToken = dis(gen); // Cryptographically secure random number  cout << "Session token: " << sessionToken << endl;  // Unpredictable and secure for authentication |

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

| **Principles(s):** MSC02-J Generates random Numbers. This principle helps emphasize a secure random number of generators which include cryptic keys and also tokens, which makes such security codes less predictable. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High(3) | Probable(2) | Low(3) | Medium P18 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S5276  <https://rules.sonarsource.com/cpp/RSPEC-5276> | This tool helps detect inefficient library algorithm usage. |
| SonarQube | 10.3+ | cpp:S6004  <https://rules.sonarsource.com/cpp/RSPEC-6004> | This tool helps detect unnecessary copy operations. |
| SonarQube | 10.3+ | java:S1319  <https://rules.sonarsource.com/java/RSPEC-1319> | This tool helps identify declarations using implementations or interfaces. |
| SonarQube | 10.3+ | java:S1149  <https://rules.sonarsource.com/java/RSPEC-1149> | This tool helps identify collection wrappers. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management with RAII | [STD-010-CPP] | RAII, Resource Acquisition is Initialization, helps ensure resources cleanup preventing leaks. |

| **Noncompliant Code** |
| --- |
| This code block shows how to manually manage resources without automatic cleanup, which in turn can lead to potential resource leaks if exceptions occur. |
| void processFile() {  FILE\* file = fopen("data.txt", "r"); // Manual resource acquisition  if (!file) return;  // Process file  char buffer[100];  fgets(buffer, 100, file);  // If exception occurs here, file never gets closed  if (buffer[0] == 'X') {  return; // Resources leak occurs due to file not closed  }  fclose(file); // Manual cleanup  } |

| **Compliant Code** |
| --- |
| This code uses RAII principles with automatic resource management which in turn ensures resources are always properly cleaned up regardless of exit. |
| void processFile() {  ifstream file("data.txt"); // RAII automatic resource management  if (!file.is\_open()) return;  // Process file  string line;  getline(file, line);  // If exception occurs, destructor closes file  if (line[0] == 'X') {  return; // No resource leak since file automatically closed  }  // File automatically closed when object goes out of scope  } |

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

| **Principles(s): MEM00-C:** Allocates and free memory in a module. This principle helps with cleanup and also prevents resource leaks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable(2( | Low(3) | Medium P12 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3+ | cpp:S1186  <https://rules.sonarsource.com/cpp/RSPEC-1186> | This tool helps detect functions that do not use documentation. |
| SonarQube | 10.3+ | cpp:S1135  <https://rules.sonarsource.com/cpp/RSPEC-1135> | This tool helps detect TO DO tags which need to be documented. |
| SonarQube | 10.3+ | java:S1186  <https://rules.sonarsource.com/java/RSPEC-1186> | This tool helps identify methods that are empty. |
| SonarQube | 10.3+ | java:S1135  <https://rules.sonarsource.com/java/RSPEC-1135> | This tool helps identify incomplete sections that use TODO or FIXME. |

### 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.

**Assess and plan:**

The security requirements are identified during the first stage. SonarQube tools can be used to check code against the requirements that were identified.

**Design: & Build:**

During development, security scanning should be integrated into the CD pipeline. Each commit triggers a check for compliance with the coding standards of STD-001-CPP to STD-010-CPP. If the processes fail if there are significant security vulnerabilities.

**DEV SEC OPS:**

Security tools will be utilized during each stage. Such as, pre commit hooks for basic security checks, servers running security scans, automated testing will include multiple test cases, and code quality hates will prevent non-compliant code.

**Monitoring/Maintaining/Detection:**

After deployment monitoring allows us to capture security metrics. This feedback can then be used to assess and plan phase.

### 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(3) | Probable(2) | Low(3) | Medium P18 | L2 |
| STD-002-CPP | High(3) | Likely(3) | Medium(2) | High P27 | L1 |
| STD-003-CPP | High(3) | Probable(2) | Low(3) | Medium P18 | L2 |
| STD-004-CPP | High(3) | Likely(3) | Medium(2) | High P27 | L1 |
| STD-005-CPP | Medium(2) | Probable(2) | Low(3) | Medium P12 | L2 |
| STD-006-CPP | High(3) | Likely(3) | High(1) | High P9 | L1 |
| STD-007-CPP | High(3) | Probable(2) | Medium(2) | Medium P18 | L2 |
| STD-008-CPP | High(3) | Likely(3) | Medium(2) | High P18 | L1 |
| STD-009-CPP | High(3) | Probable(2) | Low(3) | Medium P18 | L2 |
| STD-010-CPP | Medium(2) | Probable(2) | Low(3) | Medium P12 | L2 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 | **What:** Encryption at rest protects data in many areas, such as file systems, databases and any storage.  **How:** Any sensitive data that Green Pace, including customer records, credential, or financial records to name a few, must be encrypted using AES-256 or stronger.  **Why:** Encryption at rest prevents unauthorized access and user from accessing data. This policy ensures that data protection regulations are followed, aiding in protection against security breaches or other security threats. |
| Encryption in flight | **What:** Encryption in flight helps protect data as it moves between systems, networks, ect.  **How:** All data that is in transit between systems must use TLS.1.2. Certificates also must be validated to ensure proper security.  **Why:** Encryption in flight helps prevents man-in-the-middle or other data interception and ensures data security while in transit. |
| Encryption in use | **What:** Encryption in use helps protect data while it is being processed or applications.  **How:** For sensitive information such as payment data, personal identifiable information, or cryptographic keys, must remain encrypted during processing to ensure security.  **Why:** Encryption in use helps protect against unauthorized access of data while it is being processed, it also combats memory dumps. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **What:** Authentication helps verify the identity of users, systems, services which are trying to gain access to Green Pace data.  **How:** For logins multi-factor authentication can be used and strong passwords of certain length can be required.  **Why:** Proper authentication can prevent unauthorized access which ensure that only verified users have access to Green Pace systems. |
| Authorization | **What:** Authorizationconstraint users to certain parts of the system based on the permission they have.  **How:** This can be implemented through role-base access. By allowing users to have the least number of permissions for their job role and permissions must be reviewed frequently.  **Why:** Authorization limits the amount of data exposed to users, restricting modification or even access for certain users decreasing the amount of risk. |
| Accounting | **What:** Accounting creates logs with valuable information describing user activity or other security actions, allowing us to better monitor and analyze security actions.  **How:** This can be implemented by login authentication regardless of failed or passed. Keeping log of changes to the database is also another why to implement this policy.  **Why:** This policy helps aid in investigation regarding security incidents, and helps prove compliance. |

**\***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.

**STD-001-CPP**

**Principles**: 1. Validate Input Data, 2. Adopt a Secure Coding Standard

**Justification:** Using correct data types prevents overflow/underflow vulnerabilities that could be exploited through malicious input. This standard validates that data fits within expected ranges and enforces type safety.

**STD-002-CPP**

**Principles:** 1. Validate Input Data, 7. Sanitize Data Sent to Other Systems

**Justification:** Input validation is the primary defense against injection attacks. By validating and sanitizing all external input, this standard prevents malicious data from compromising system integrity.

**STD-003-CPP**

**Principles:** 1. Validate Input Data, 4. Keep It Simple

**Justification:** Proper string handling with boundary checks prevents buffer overflows while maintaining code simplicity. Safe string functions reduce complexity compared to manual boundary management.

**STD-004-CPP**

**Principles:** 1. Validate Input Data, 7. Sanitize Data Sent to Other Systems, 8. Practice Defense in Depth

**Justification:** Parameterized queries prevent SQL injection by treating user input as data rather than executable code. This provides a critical security layer protecting database integrity.

**STD-005-CPP**

**Principles:** 6. Adhere to the Principle of Least Privilege, 8. Practice Defense in Depth

**Justification:** Proper memory management prevents unauthorized access to freed memory and reduces attack surface. It ensures resources are available only when needed and properly cleaned up.

**STD-006-CPP**

**Principles:** 2. Heed Compiler Warnings, 9. Use Effective Quality Assurance Techniques

**Justification:** Assertions catch violations of assumptions during development and testing, serving as quality gates that identify defects before production deployment.

**STD-007-CPP**

**Principles:** 8. Practice Defense in Depth, 4. Keep It Simple

**Justification:** Exception handling provides a safety layer that prevents crashes and information disclosure while maintaining simple, readable error management code.

**STD-008-CPP**

**Principles:** 1. Validate Input Data, 6. Adhere to the Principle of Least Privilege

**Justification:** Path validation prevents traversal attacks by restricting file access to authorized directories only, implementing least privilege for file system operations.

**STD-009-CPP**

**Principles:** 3. Architect and Design for Security Policies, 10. Adopt a Secure Coding Standard

**Justification:** Cryptographically secure random generation is essential for security architecture, ensuring unpredictability in tokens, keys, and security-sensitive operations.

**STD-010-CPP**

**Principles:** 4. Keep It Simple, 8. Practice Defense in Depth

**Justification:** RAII simplifies resource management while providing automatic cleanup as a defense layer, ensuring resources are properly managed even during exceptional conditions.

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.2 | 10/4/2025 | 10 Principles | Christina Jimenez |  |
| 1.3 | 10/10/2025 | Risks Assessments | Christina Jimenez |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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