

# 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 data ensures that all data coming from outside the system is checked for correctness and appropriateness before being processed. This helps prevent common vulnerabilities such as SQL injection, buffer overflow, and cross-site scripting (XSS) by ensuring that the input conforms to expected formats and ranges. |
| 1. Heed Compiler Warnings | Compiler warnings can indicate potential issues in the code that could lead to security vulnerabilities. By addressing these warnings, developers can catch and fix issues early in the development process, reducing the risk of security flaws and improving the overall robustness of the code. |
| 1. Architect and Design for Security Policies | Security should be considered at the architectural and design stages of software development. By incorporating security policies and principles early on, developers can create systems that are inherently more secure and resilient to attacks. This proactive approach helps in identifying potential security issues before they become problematic. |
| 1. Keep It Simple | Simple and straightforward designs are easier to understand, implement, and review for security flaws. Complexity can obscure potential vulnerabilities and make the code more difficult to maintain and secure. Keeping designs and implementations simple minimizes the risk of introducing hidden security issues. |
| 1. Default Deny | The default deny principle states that, by default, access should be denied to resources and actions unless explicitly allowed. This minimizes the risk of unauthorized access and ensures that only explicitly granted permissions are given, thus reducing the attack surface. |
| 1. Adhere to the Principle of Least Privilege | Each component of a system should operate with the minimum privileges necessary to perform its tasks. By adhering to this principle, the impact of a security breach can be minimized, as attackers will have limited access and control over the system if they compromise a component. |
| 1. Sanitize Data Sent to Other Systems | When data is sent from one system to another, it should be sanitized to ensure that it does not contain malicious code or invalid content. This prevents the introduction of vulnerabilities into other systems and ensures data integrity and security across different platforms. |
| 1. Practice Defense in Depth | Defense in depth involves implementing multiple layers of security controls and measures throughout a system. This layered approach ensures that if one security measure fails, others are in place to provide additional protection, making it harder for attackers to penetrate the system. |
| 1. Use Effective Quality Assurance Techniques | Implementing thorough quality assurance practices, including code reviews, testing, and static analysis, helps in identifying and fixing security vulnerabilities early in the development process. Effective QA techniques ensure that the code meets security standards and is free of common security flaws. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard, such as the SEI CERT C++ Coding Standard, ensures that code is written with security in mind. These standards provide guidelines and best practices for avoiding common security vulnerabilities and producing robust, secure software. |

### 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 Safety** |
| --- | --- | --- |
| **Data Type** | STD-001-C++ | Ensuring data type safety is crucial in preventing type-related vulnerabilities, such as buffer overflows and type confusion. This standard enforces correct and consistent usage of data types to ensure the integrity and security of the program. |

| **Noncompliant Code** |
| --- |
| This code allocates memory for characters but assigns it to an integer pointer, which can cause undefined behavior and security issues. |
| int \*ptr;  ptr = (int \*)malloc(sizeof(char) \* 10); // Incorrect memory allocation |

| **Compliant Code** |
| --- |
| This code correctly allocates memory for integers, ensuring type safety and preventing potential vulnerabilities. |
| int \*ptr;  ptr = (int \*)malloc(sizeof(int) \* 10); // Correct memory allocation |

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

| **Principle: Secure Coding Practices:**  This principle emphasizes the importance of preventing security vulnerabilities through careful and consistent use of data types. By enforcing strict data type safety, the standard ensures that type-related issues, such as buffer overflows and type confusion, are avoided, which aligns with the secure coding principle of minimizing attack vectors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Cppcheck** | 2.10 | memleak | Cppcheck is an open-source static analysis tool that detects memory leaks, data type mismanagement, and other code issues in C++. The memleak checker specifically helps identify memory allocation problems that could arise from improper data type usage. |
| **Clang-Tidy** | 14.0.0 | modernize-use-auto | Clang-Tidy is a part of the LLVM project and offers a wide range of checks to enforce coding standards. The modernize-use-auto checker helps ensure that the correct data types are used, reducing the risk of type-related issues. |
| **SonarQube** | 9.9 | S1862 (Correctness of memory allocation) | SonarQube is a comprehensive code quality tool that provides static analysis across multiple languages, including C++. The S1862 rule ensures that memory allocation functions are used correctly, preventing type-related vulnerabilities. |
| **PVS-Studio** | 7.19 | V512 (Pointers to different types) | PVS-Studio is a static code analyzer that identifies bugs and security weaknesses in C++ code. The V512 checker helps identify cases where pointers to different types are used incorrectly, which can lead to undefined behavior. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Validation** |
| --- | --- | --- |
| **Data Value** | STD-002-C++ | Validating data values before use ensures that they meet expected criteria, preventing erroneous behavior and potential security vulnerabilities caused by invalid data. |

| **Noncompliant Code** |
| --- |
| This code does not check for negative values, leading to incorrect calculations. |
| int calculate\_area(int width, int height) {  return width \* height;  }  int main() {  int area = calculate\_area(-5, 10); // No validation for negative values  printf("Area: %d\n", area);  return 0;  } |

| **Compliant Code** |
| --- |
| This code validates the input values to ensure they are non-negative before performing calculations. |
| int calculate\_area(int width, int height) {  if (width < 0 || height < 0) {  printf("Invalid dimensions\n");  return -1;  }  return width \* height;  }  int main() {  int area = calculate\_area(5, 10); // Validated input values  printf("Area: %d\n", area);  return 0;  } |

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

| **Principle: Input Validation and Data Integrity**  This principle underscores the importance of validating input data to ensure it conforms to expected criteria. By validating data values before use, the standard helps prevent erroneous behavior and potential security vulnerabilities that could arise from invalid data, thus ensuring the integrity and reliability of the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Cppcheck** | 2.10 | knownConditionTrueFalse | Cppcheck’s knownConditionTrueFalse checker identifies instances where assumptions about input values may be incorrect, helping to catch cases where validation might be missing. |
| **Clang-Tidy** | 14.0.0 | readability-const-return-type | Clang-Tidy offers a variety of checks including those that ensure functions are properly handling and validating inputs to avoid potential issues. |
| **SonarQube** | 9.9 | S1166 (Unused function parameters) | SonarQube can identify functions that are not properly handling or validating parameters, highlighting where additional validation might be needed. |
| **PVS-Studio** | 7.19 | V601 (The function can return an incorrect result) | PVS-Studio’s V601 checker helps detect functions that might return incorrect results due to improper validation of input values. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness** |
| --- | --- | --- |
| **String Correctness** | STD-003-C++ | Ensuring string correctness prevents buffer overflows and other vulnerabilities associated with improper handling of strings. This standard enforces proper management and validation of string data. |

| **Noncompliant Code** |
| --- |
| This code does not check the length of the input string, leading to potential buffer overflow. |
| void copy\_string(char \*dest, const char \*src) {  strcpy(dest, src); // No length check  }  int main() {  char buffer[10];  copy\_string(buffer, "This is a very long string"); // Potential buffer overflow  printf("Buffer: %s\n", buffer);  return 0;  } |

| **Compliant Code** |
| --- |
| This code checks the length of the input string and uses a safe copy function to prevent buffer overflow. |
| void copy\_string(char \*dest, const char \*src, size\_t dest\_size) {  strncpy(dest, src, dest\_size - 1);  dest[dest\_size - 1] = '\0'; // Ensure null-termination  }  int main() {  char buffer[10];  copy\_string(buffer, "This is a very long string", sizeof(buffer)); // Safe copy  printf("Buffer: %s\n", buffer);  return 0;  } |

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

| **Principle: Proper String Management**  This principle emphasizes the importance of managing and validating string data to prevent common vulnerabilities such as buffer overflows. By ensuring strings are handled correctly, this standard helps protect against security risks associated with improper string operations, such as memory corruption and unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Cppcheck** | 2.10 | bufferOverrun | Cppcheck’s bufferOverrun checker helps identify potential buffer overflows by analyzing string handling code, ensuring that string operations are performed safely. |
| **Clang-Tidy** | 14.0.0 | performance-unnecessary-copy-initialization | Clang-Tidy can detect unnecessary copies and help ensure that string handling operations are optimized and secure. |
| **SonarQube** | 9.9 | S2270 (Use of strcpy/strcat functions) | SonarQube identifies unsafe usage of string functions like strcpy and strcat, recommending safer alternatives to prevent buffer overflow vulnerabilities. |
| **PVS-Studio** | 7.19 | V579 (The function can be unsafe) | PVS-Studio’s V579 checker helps detect functions that may be unsafe due to improper handling of string lengths, highlighting potential buffer overflow issues. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Prevention** |
| --- | --- | --- |
| **SQL Injection** | STD-004-C++ | SQL Injection is a major security vulnerability that allows attackers to interfere with the queries that an application makes to its database. This standard enforces practices that prevent SQL Injection attacks by using parameterized queries and proper input validation. |

| **Noncompliant Code** |
| --- |
| This code constructs an SQL query using string concatenation, which is vulnerable to SQL Injection. |
| void executeQuery(const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'";  // Execute query  } |

| **Compliant Code** |
| --- |
| This code uses parameterized queries to prevent SQL Injection. |
| void executeQuery(const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE name = ?";  // Use a prepared statement to set the parameter and execute the query  } |

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

| **Principle: Secure Query Handling**  This principle underscores the importance of preventing SQL Injection by ensuring that all database queries are constructed in a secure manner. By using parameterized queries and proper input validation, this standard helps to mitigate the risk of SQL Injection attacks, thereby protecting the database and application from unauthorized access and manipulation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **SonarQube** | 9.9 | S5142 (SQL Injection) | SonarQube’s SQL Injection checker detects potential SQL Injection vulnerabilities in code, helping to identify insecure query construction practices. |
| **Checkmarx** | 9.3 | SQL Injection Detection | Checkmarx provides static code analysis to identify SQL Injection vulnerabilities and ensure that queries are parameterized properly. |
| **Fortify Static Code Analyzer** | 22.1 | SQL Injection | Fortify’s SQL Injection checker analyzes code for insecure database queries and provides recommendations for using parameterized queries. |
| **Veracode** | 2024.1 | SQL Injection | Veracode’s security scanning tool identifies potential SQL Injection vulnerabilities by examining query construction and data handling practices. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection** |
| --- | --- | --- |
| **Memory Protection** | STD-005-C++ | Memory protection ensures that applications handle memory operations safely, preventing issues such as buffer overflows, use-after-free errors, and memory leaks. This standard enforces best practices for dynamic memory management. |

| **Noncompliant Code** |
| --- |
| This code does not check if the memory allocation was successful, leading to potential null pointer dereference. |
| int \*allocateArray(int size) {  int \*array = (int \*)malloc(size \* sizeof(int));  // Use array without checking if allocation was successful  array[0] = 1;  return array;  } |

| **Compliant Code** |
| --- |
| This code checks if the memory allocation was successful before using the allocated memory. |
| int \*allocateArray(int size) {  int \*array = (int \*)malloc(size \* sizeof(int));  if (array == NULL) {  // Handle memory allocation failure  return NULL;  }  array[0] = 1;  return array;  } |

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

| **Principle: Safe Memory Management** This principle emphasizes the importance of verifying successful memory allocation and proper handling of memory. By ensuring that memory operations are checked and managed correctly, this standard helps prevent common issues such as null pointer dereferences and memory leaks. |
| --- |

**Threat Level**

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

**Automation**

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Valgrind | 3.20.1 | Memory Leak Detection | Helps in detecting memory leaks and improper memory management. |
| AddressSanitizer | 15.0 | Buffer Overflow and Use-After-Free Detection | Detects memory errors such as buffer overflows and use-after-free issues. |
| Dr. Memory | 2.3 | Memory Error Detection | Detects various memory errors, including leaks and invalid memory accesses. |
| Coverity | 2024.1 | Memory Management | Provides static code analysis for detecting memory management issues and potential vulnerabilities. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-C++ | Assertions are used to catch programming errors by verifying assumptions made by the code during runtime. This standard enforces the use of assertions to ensure that critical assumptions hold true, helping to identify logic errors and potential vulnerabilities early. |

| **Noncompliant Code** |
| --- |
| This code does not use assertions to check for valid input, potentially leading to undefined behavior. |
| int divide(int a, int b) {  return a / b; // No check for division by zero  } |

| **Compliant Code** |
| --- |
| This code uses assertions to ensure that the divisor is not zero before performing the division. |
| #include <cassert>  int divide(int a, int b) {  assert(b != 0); // Check for division by zero  return a / b;  } |

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

| **Principle: Error Detection**  This principle highlights the use of assertions to verify critical assumptions during runtime. Assertions help in early identification of logic errors and potential vulnerabilities by ensuring that the program state matches expected conditions. |
| --- |

**Threat Level**

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

**Automation**

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| GDB | 12.2 | Runtime Assertion Checking | Provides runtime support for assertions and helps in debugging failed assertions. |
| Clang | 16.0 | Static Analysis | Includes assertion checking in static code analysis to identify potential issues. |
| Coverity | 2024.1 | Static Code Analysis | Detects logical errors and issues related to assertions through static code analysis. |
| Cppcheck | 2.8 | Static Code Analysis | Provides checks for assertions and other logical issues in C++ code. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-C++ | Proper use of exceptions helps manage errors and exceptional conditions in a controlled manner, ensuring that the program can handle unexpected situations gracefully. This standard enforces best practices for throwing, catching, and handling exceptions. |

| **Noncompliant Code** |
| --- |
| This code does not handle exceptions, leading to potential program crashes. |
| void readFile(const std::string& filename) {  std::ifstream file(filename);  // No exception handling  file >> data;  } |

| **Compliant Code** |
| --- |
| This code uses try-catch blocks to handle exceptions and ensure that the program can manage errors gracefully. |
| void readFile(const std::string& filename) {  try {  std::ifstream file(filename);  if (!file) {  throw std::runtime\_error("File not found");  }  file >> data;  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  } |

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

| **Principle: Error Handling**  This principle emphasizes the importance of managing errors and exceptional conditions through structured exception handling. Proper use of exceptions ensures that errors are handled gracefully, preventing program crashes and maintaining robust functionality. |
| --- |

**Threat Level**

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

**Automation**

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Valgrind | 3.19.0 | Memory Error Detection | Detects issues related to memory and error handling. |
| Clang | 16.0 | Static Analysis | Identifies potential exceptions-related issues in code. |
| Coverity | 2024.1 | Static Code Analysis | Provides insights into error handling and exception management. |
| Cppcheck | 2.8 | Static Code Analysis | Checks for proper exception handling and error management in C++ code. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Safe Integer Operations** |
| --- | --- | --- |
| Safe Integer Operations | STD-008-C++ | Integer overflows and underflows can lead to serious security vulnerabilities, including arbitrary code execution and denial-of-service attacks. This standard enforces practices to ensure safe arithmetic operations on integers, preventing these types of vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code performs arithmetic operations without checking for overflow, leading to potential undefined behavior. |
| int multiply(int a, int b) {  return a \* b; // No check for overflow  } |

| **Compliant Code** |
| --- |
| This code uses built-in functions to check for overflow during arithmetic operations. |
| #include <limits>  #include <stdexcept>  int multiply(int a, int b) {  if (a > 0 && b > 0 && a > (std::numeric\_limits<int>::max() / b)) {  throw std::overflow\_error("Integer overflow");  }  return a \* b;  } |

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

| **Principles: Fail-Safe Defaults** The Fail-Safe Defaults principle suggests that systems should be designed to fail in a safe manner when encountering unexpected conditions. In the context of safe integer operations, this principle implies that arithmetic operations involving integers should include checks to prevent overflows and underflows, ensuring that if an operation is about to exceed the capacity of the data type, the system will handle it gracefully, either by throwing an exception or managing the error appropriately. |
| --- |

**Threat Level**

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

**Automation**

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0.0 | -analyze | Checks for integer overflows and other issues. |
| Coverity | 2024.1 | Integer Overflow Checker | Detects potential integer overflow during static analysis. |
| CodeSonar | 10.1 | Buffer Overflow Checker | Identifies unsafe arithmetic operations. |
| Fortify Static Code Analyzer | 21.2 | Data Flow Analysis | Analyzes potential vulnerabilities in integer operations. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Secure File Handling** |
| --- | --- | --- |
| Secure File Handling | STD-009-C++ | Improper file handling can lead to various security vulnerabilities, such as race conditions, file tampering, and unauthorized access. This standard enforces best practices for securely handling files, including proper validation, access controls, and error checking. |

| **Noncompliant Code** |
| --- |
| This code opens a file without proper validation or error checking. |
| void readFile(const std::string& filename) {  std::ifstream file(filename);  // No validation or error checking  file >> data;  } |

| **Compliant Code** |
| --- |
| This code validates the filename and checks for errors when opening the file. |
| #include <stdexcept>  void readFile(const std::string& filename) {  if (filename.empty()) {  throw std::invalid\_argument("Filename cannot be empty");  }  std::ifstream file(filename);  if (!file.is\_open()) {  throw std::runtime\_error("Could not open file");  }  file >> data;  } |

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

| **Principles(s):**  Principle of Least Privilege: Ensures that files are accessed only when necessary and with the least amount of privilege, reducing potential exposure to unauthorized access or tampering.  Fail-Safe Defaults: Ensures that proper error handling is in place to handle scenarios where file operations might fail, thereby avoiding undefined behavior or potential security risks. |
| --- |

**Threat Level**

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

**Automation**

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0.0 | -analyze | Detects improper file handling and potential issues. |
| Coverity | 2024.1 | File Handling Checker | Identifies potential vulnerabilities in file operations. |
| CodeSonar | 10.1 | File Access Checker | Analyzes file handling practices for security issues. |
| Fortify Static Code Analyzer | 21.2 | File Security Analysis | Provides insights into secure file handling and practices. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Secure Network Communication** |
| --- | --- | --- |
| Secure Network Communication | STD-010-C++ | Secure network communication ensures that data transmitted over the network is protected from interception, tampering, and unauthorized access. This standard enforces practices such as encryption, secure protocols, and proper validation to secure network communication. |

| **Noncompliant Code** |
| --- |
| This code transmits data over the network without encryption, making it vulnerable to interception. |
| void sendData(const std::string& data) {  // Transmit data without encryption  network.send(data);  } |

| **Compliant Code** |
| --- |
| This code uses SSL/TLS to encrypt data before transmitting it over the network. |
| #include <openssl/ssl.h>  #include <openssl/err.h>  void sendData(const std::string& data) {  SSL \*ssl = /\* Initialize SSL \*/;  SSL\_write(ssl, data.c\_str(), data.length());  // Properly handle SSL cleanup and error checking  } |

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

| **Principles(s):**  Confidentiality: Ensures that data transmitted over the network is encrypted to prevent unauthorized access and interception, maintaining the confidentiality of the information.  Integrity: Guarantees that data remains intact and unaltered during transmission by using secure protocols and encryption, protecting against tampering.  Authentication: Verifies the identity of the parties involved in the communication to ensure that data is sent and received by legitimate entities. |
| --- |

**Threat Level**

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| Critical | High | High | High | Very High |

**Automation**

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| OpenSSL | 3.0.0 | SSL/TLS Security Checker | Checks for proper SSL/TLS usage and encryption. |
| Fortify Static Code Analyzer | 21.2 | Network Security Analysis | Analyzes secure communication practices. |
| Checkmarx | 8.0 | Secure Transmission Checker | Detects vulnerabilities in network communication. |
| Veracode | 2024.1 | Encryption Validation | Validates proper use of encryption protocols. |

### 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 is essential for enforcing the security standards in this policy. Green Pace's existing DevOps infrastructure will be enhanced by integrating static code analysis tools into our CI pipeline, like SonarQube, and security testing tools in our CD pipeline, such as OWASP ZAP. Configuration management tools like Chef will automate security settings, and continuous monitoring will be supported by automated solutions for real-time visibility and compliance. These updates will ensure efficient and consistent adherence to security standards throughout our development process.

### 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-C++ | High | Unlikely | Medium | High | 2 |
| STD-002-C++ | High | Moderate | Low | High | Mandatory |
| STD-003-C++ | High | Moderate | Moderate | High | Mandatory |
| STD-004-C++ | Critical | High | High | Critical | Mandatory |
| STD-005-C++ | High | Medium | Medium | High | Important |
| STD-006-C++ | Medium | Low | Low | Medium | Moderate |
| STD-007-C++ | High | Medium | Medium | High | Critical |
| STD-008-C++ | High | Medium | Low | High | Critical |
| STD-009-C++ | High | Medium | Moderate | High | Critical |
| STD-010-C++ | Critical | High | High | High | Very High |

### 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** | **Explanation:** Encryption at rest refers to encrypting data stored on physical media, such as hard drives or databases. This protects data from unauthorized access when it is not actively being used. **Application:** This policy applies to all stored data, including backups and archives. It is crucial for protecting against data breaches and theft, ensuring compliance with data protection regulations like GDPR or HIPAA. |
| **Encryption in Flight** | **Explanation:** Encryption in flight protects data as it travels across networks, including the internet and internal company networks. This encryption safeguards data from interception or tampering during transmission. **Application:** This policy applies whenever data is transmitted. It ensures that data remains secure during communication, preventing unauthorized access or alteration, and is essential for maintaining data integrity and confidentiality. |
| **Encryption in Use** | **Explanation:** Encryption in use secures data while it is being actively processed or accessed by applications. This protects data from unauthorized exposure or manipulation during active use. **Application:**This policy applies to scenarios involving data processing or computation. It maintains data privacy and security throughout its lifecycle, ensuring protection against unauthorized access during active operations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| **Authentication** | **Explanation:** Authentication verifies the identity of users or systems attempting to access resources. It ensures that only legitimate users are granted access based on their credentials, such as usernames, passwords, or biometric data. **Application:** This policy applies to all systems requiring user access, including login processes and secure transactions. It is crucial for preventing unauthorized access and ensuring that only verified entities can interact with sensitive data. |
| **Authorization** | **Explanation:** Authorization determines the permissions and access levels granted to authenticated users. It ensures that users can only access the resources and perform actions that their role or permissions allow. **Application:** This policy applies after authentication, governing access to various system features and data based on user roles or attributes. It prevents privilege escalation and ensures users can only interact with resources within their designated scope. |
| **Accounting** | **Explanation:** Accounting involves tracking and logging user activities and resource usage. It provides a record of who accessed what data, when, and what actions were performed. **Application:** This policy applies to all systems where tracking of user actions is necessary, including database interactions and system changes. It is essential for auditing, detecting anomalies, and maintaining accountability, thereby supporting forensic investigations and compliance with regulatory requirements. |

**\***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.

**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 |  |
| 3.0 | 08/10/2024 | Completed Template | Elijah Paulk |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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