**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 | Ensuring that input data is validated prevents attackers from injecting malicious data into systems. Input validation checks that the data received by a system is what it expects, mitigating risks like SQL injection and buffer overflows. |
| 1. Heed Compiler Warnings | Compiler warnings often flag potential security issues like memory leaks, uninitialized variables, or type mismatches. Addressing these warnings can help identify vulnerabilities early in the development process, reducing the risk of security flaws. |
| 1. Architect and Design for Security Policies | Secure architecture and design involve planning security into the system from the beginning. This includes many things, including defining security policies, access control mechanisms, and ensuring components are secure and isolated, following security principles such as least privilege. |
| 1. Keep It Simple | Complex systems are more likely to contain security vulnerabilities than simpler systems. By simplifying the design and code of a system, you reduce the risk of hidden bugs and make the system easier to audit and maintain. |
| 1. Default Deny | The principle of default deny ensures that all access is denied by default, and only explicitly allowed actions are permitted. This reduces the attack surface by preventing unauthorized actions from being executed. |
| 1. Adhere to the Principle of Least Privilege | Users and systems should only have the minimum level of access necessary to perform their tasks. By limiting privileges, you reduce the potential damage from security breaches. |
| 1. Sanitize Data Sent to Other Systems | Data passed between systems should be sanitized to ensure no malicious content is transferred. This helps prevent attacks such as cross-site scripting and SQL injection when data is transferred. |
| 1. Practice Defense in Depth | Defense in depth involves implementing multiple layers of security, so if one fails, others are still in place to provide protection. This principle ensures that no single point of failure compromises the entire system. |
| 1. Use Effective Quality Assurance Techniques | Regularly testing and auditing code with techniques like static analysis, dynamic testing, and penetration testing helps identify and fix security vulnerabilities before they can be exploited. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard ensures that code is developed with the best security practices in mind. These standards provide guidelines on how to write code that is less vulnerable to common security threats. |

### 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 | Ensure that data types are used correctly to prevent unexpected behavior or vulnerabilities due to incorrect type handling. |

| **Noncompliant Code** |
| --- |
| In this example, incorrect data types are used for variables, which may lead to unintended results or security vulnerabilities. |
| int x = 10.5; // implicit conversion from double to int.  char c = 300; // overflow. 300 exceeds char limit. |

| **Compliant Code** |
| --- |
| The correct data types are used here, ensuring no data loss or overflow, improving reliability and safety. |
| double x = 10.5; // correct type for floating-point value.  int c = 300; // correct type for value in range. |

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

| **Principles(s):**  1: **Validate Input Data** - Correctly using data types prevents improper input handling, reducing the risk of buffer overflows and other vulnerabilities.  2: **Heed Compiler Warnings** - Compiler warnings often flag potential issues with type conversions, ensuring they are addressed early in the development process.  8: **Practice Defense in Depth** - Using strict data type enforcement acts as one layer of defense in the overall security strategy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | UndefinedBehaviorSanitizer | Detects potential issues related to type conversions and overflow. |
| SonarQube | 9.3 | C | Identifies non-compliant type usage and overflow vulnerabilities. |
| Coverity | 2024.1 | CERT.CERT-FLP34-C | Checks for floating-point value and type conversion vulnerabilities. |
| Cppcheck | 2.8 | Type conversion | Analyzes code for unsafe type conversions and implicit data type changes. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Safeguards against improper manipulation or assignment of data values, which could lead to overflows, underflows, or security breaches. |

| **Noncompliant Code** |
| --- |
| This code does not check the bounds of the data, leading to overflow or underflow issues. |
| int a = INT\_MAX; // Maximum value for an int.  a += 1; // Causes overflow. |

| **Compliant Code** |
| --- |
| Here, checks are added to ensure the data value stays within safe limits, avoiding overflow. |
| int a = INT\_MAX;  if (a < INT\_MAX) {  a += 1; // Only increment if it is safe  } else {  // Handle case of overflow  } |

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

| **Principles(s):**  1: **Validate Input Data** - This principle supports verifying that the data being manipulated stays within valid ranges, preventing overflow or underflow situations.  8: **Practice Defense in Depth** - Implementing checks for data values is a defense mechanism against unexpected data manipulation, adding another layer of security.  9: **Use Effective Quality Assurance Techniques** - Regular testing can identify instances where data manipulation may lead to overflow, ensuring that safeguards are effectively applied. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | Integer Overflow | Detects potential integer overflow and underflow situations in code. |
| SonarQube | 9.3 | C | Identifies instances where unsafe data value manipulation can lead to overflows. |
| Coverity | 2024.1 | CERT.INT30-C | Checks for integer overflow and underflow in data assignments and manipulations. |
| Cppcheck | 2.8 | Integer Handling | Analyzes code to detect possible overflow, underflow, and unsafe integer operations. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Prevents issues like buffer overflow or injection attacks caused by improper handling of strings. |

| **Noncompliant Code** |
| --- |
| The code does not properly terminate a string, potentially causing buffer overflows or undefined behavior. |
| char str[5];  strcpy(str, "Hello, world!"); // Buffer overflow. Array is too small |

| **Compliant Code** |
| --- |
| The code properly handles the string, ensuring no overflow by limiting the number of characters copied. |
| char str[5];  strncpy(str, "Hello, world!", sizeof(str) - 1); // Copy only up to the buffer size  str[4] = '\0'; // Ensure null-termination |

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

| **Principles(s):**  1: **Validate Input Data** - Properly handling strings involves validating and restricting the input size to prevent buffer overflows.  4: **Keep It Simple** - Using safe string-handling functions simplifies the code, reducing the risk of errors that could lead to vulnerabilities.  8: **Practice Defense in Depth** - String checks are an additional layer of security to defend against potential attacks like buffer overflows and injection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | BufferOverflow | Detects possible buffer overflow conditions caused by improper string handling. |
| SonarQube | 9.3 | C | Identifies instances of insecure string handling that may lead to overflows. |
| Coverity | 2024.1 | CERT.STR31-C | Checks for common string manipulation issues, including overflow vulnerabilities. |
| Cppcheck | 2.8 | String Handling | Analyzes code for potential buffer overflow and unsafe string operations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Mitigates risks associated with unvalidated input in SQL queries, which could allow attackers to manipulate the database. |

| **Noncompliant Code** |
| --- |
| This example shows vulnerable code that directly uses unsanitized user input in SQL queries. |
| std::string userInput = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  executeQuery(query); // Vulnerable to SQL injection attacks |

| **Compliant Code** |
| --- |
| The code uses prepared statements to prevent SQL injection attacks. |
| std::string userInput = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement stmt = connection.prepareStatement(query);  stmt.setString(1, userInput);  executeQuery(stmt); // Safe against SQL injection |

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

| **Principles(s):**  1: **Validate Input Data** - Using prepared statements ensures that input data is validated and treated as a parameter, not as executable code, thus preventing SQL injection attacks.  6: **Adhere to the Principle of Least Privilege** - By controlling how inputs interact with SQL queries, you minimize the privileges required to access the database, reducing potential misuse.  8: **Practice Defense in Depth** - Employing prepared statements and input validation provides an additional security layer to prevent SQL injections even if other defenses are compromised. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | C | Identifies instances of SQL queries constructed using user inputs without using prepared statements. |
| Coverity | 2024.1 | CERT.SQL33-C | Checks for the correct use of parameterized queries to prevent SQL injection. |
| Fortify Static Code Analyzer | 20.2 | SQL Injection | Scans code for SQL injection vulnerabilities and recommends secure coding practices. |
| Clang Static Analyzer | 12.0.1 | SQL Injection | Analyzes code to identify unsafe SQL query constructions and provides guidance for remediation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Protects the system from memory corruption issues such as buffer overflows or dangling pointers. |

| **Noncompliant Code** |
| --- |
| In this code, memory is allocated but not deallocated, leading to a memory leak. |
| int\* ptr = new int[100]; // Memory allocated, but not freed |

| **Compliant Code** |
| --- |
| Memory is properly allocated and then freed, ensuring no memory leaks. |
| int\* ptr = new int[100];  delete[] ptr; // Properly deallocated memory |

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

| **Principles(s):**  2: **Heed Compiler Warnings** - Many compilers provide warnings about memory issues, such as potential memory leaks. Addressing these warnings helps ensure proper memory management.  8: **Practice Defense in Depth** - Implementing safe memory management practices, such as freeing allocated memory, is a layer of defense against memory corruption and related vulnerabilities.  9: **Use Effective Quality Assurance Techniques** - Regular testing, including memory analysis tools, helps detect memory leaks and ensures that memory is managed safely and correctly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | MemoryLeak | Detects potential memory leaks in dynamically allocated memory. |
| Valgrind | 3.17.0 | Memcheck | Identifies memory leaks, buffer overflows, and improper memory deallocations. |
| Coverity | 2024.1 | CERT.MEM31-C | Checks for correct allocation and deallocation of memory to prevent leaks and dangling pointers. |
| Cppcheck | 2.8 | Memory Leak | Analyzes code for memory leaks and improper memory deallocation. |
| AddressSanitizer | 12.0.1 | Memory Bugs | Detects memory leaks, use-after-free, and buffer overflow issues at runtime. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Ensures that assumptions in code are validated during runtime to catch potential bugs or logic errors early. |

| **Noncompliant Code** |
| --- |
| In this example, an assertion is misused, checking a condition that may be false during normal operation, leading to unnecessary program termination in production. |
| int divide(int a, int b) {  assert(b != 0); // This will terminate the program if b is 0  return a / b;  } |

| **Compliant Code** |
| --- |
| Assertions should only be used for debugging purposes, not to handle runtime errors. A proper error-handling mechanism is used here instead. |
| int divide(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Division by zero"); // Proper error handling  }  return a / b;  } |

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

| **Principles(s):**  4: **Keep It Simple** - Assertions should only be used for debugging to simplify code logic. Using assertions properly prevents unnecessary complexity in error handling during runtime.  7: **Sanitize Data Sent to Other Systems** - By ensuring assumptions (such as valid input) are checked, you can prevent invalid or dangerous data from propagating further into the system.  9: **Use Effective Quality Assurance Techniques** - Assertions are a key part of debugging and quality assurance practices, helping identify potential logic errors during the development phase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | Assert Usage | Detects potential misuse of assertions for runtime error handling. |
| SonarQube | 9.3 | C | Identifies improper use of assertions for runtime condition checks. |
| Coverity | 2024.1 | CERT.ERR33-C | Checks for inappropriate use of assertions and ensures proper error-handling mechanisms are in place. |
| Cppcheck | 2.8 | Assertion Handling | Analyzes code for assertions used in production, suggesting proper error handling instead. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handles exceptional conditions in code gracefully without causing the program to crash or enter an unstable state. |

| **Noncompliant Code** |
| --- |
| The exception is caught but not handled properly, leading to program termination or undefined behavior. |
| try {  int result = divide(10, 0);  } catch (...) {  // Catching all exceptions without handling  std::cerr << "An error occurred" << std::endl;  // Program continues in an unknown state  } |

| **Compliant Code** |
| --- |
| Exceptions are caught and handled appropriately, with specific error handling logic for different types of exceptions. |
| try {  int result = divide(10, 0);  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  // Handle specific exception  } catch (const std::exception& e) {  std::cerr << "General error: " << e.what() << std::endl;  } |

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

| **Principles(s):**  4: **Keep It Simple** - By handling exceptions properly, the code becomes easier to understand and maintain, ensuring that errors are managed in a predictable way.  8: **Practice Defense in Depth** - Proper exception handling provides an additional layer of defense against unforeseen errors, preventing unexpected program states or crashes.  9: **Use Effective Quality Assurance Techniques** - Testing exception handling as part of the quality assurance process ensures that the application behaves reliably under erroneous conditions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | Exception Handling | Identifies potential misuse of exceptions and improper error-handling patterns. |
| SonarQube | 9.3 | C | Detects instances where exceptions are caught but not handled appropriately. |
| Coverity | 2024.1 | CERT.ERR09-C | Checks for proper use of exception handling mechanisms, ensuring that all caught exceptions are managed correctly. |
| Cppcheck | 2.8 | Exception Safety | Analyzes code for correct exception handling practices and suggests improvements. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Validation** | STD-008-CPP | Ensures that all inputs to a system are properly validated before processing, which helps prevent vulnerabilities like buffer overflow, SQL injection, and command injection. |

| **Noncompliant Code** |
| --- |
| In this case, user input is not validated, allowing unsafe data to enter the system, which could lead to a buffer overflow or other vulnerabilities. |
| char buffer[10];  std::cin >> buffer; // No size check. Can lead to buffer overflow |

| **Compliant Code** |
| --- |
| The input is validated, ensuring that the buffer size is not exceeded, preventing potential overflow. |
| char buffer[10];  std::cin.get(buffer, sizeof(buffer)); // Limits input size, preventing buffer overflow |

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

| **Principles(s):**  1: **Validate Input Data** - This principle is directly related to input validation, ensuring that all data entering the system is checked and constrained to safe limits to prevent overflow and other vulnerabilities.  8: **Practice Defense in Depth** - Implementing input validation is a critical layer in the overall defense strategy, protecting against various attacks such as buffer overflows and injection.  9: **Use Effective Quality Assurance Techniques** - Rigorous testing of input validation mechanisms helps identify and fix potential vulnerabilities before they can be exploited. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | Input Validation | Detects potential input validation issues, such as unchecked buffer sizes. |
| SonarQube | 9.3 | C | Identifies cases where input is not properly validated, increasing the risk of buffer overflows. |
| Coverity | 2024.1 | CERT.STR50-C | Checks for proper validation of input data to prevent buffer overflow and other input-related vulnerabilities. |
| Cppcheck | 2.8 | Input Handling | Analyzes code for input validation errors, such as missing size checks for buffers. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Resource Management** | STD-009-CPP | Manages memory and system resources efficiently to prevent memory leaks, resource exhaustion, and related security issues. |

| **Noncompliant Code** |
| --- |
| This example shows improper resource management, where a file is opened but not closed, leading to a resource leak. |
| std::fstream file("example.txt", std::ios::in); // File is not closed, causing resource leak |

| **Compliant Code** |
| --- |
| Here, the file is properly managed using RAII (Resource Acquisition Is Initialization) with a scoped resource that automatically closes the file. |
| {  std::fstream file("example.txt", std::ios::in);  // File is automatically closed when it goes out of scope  } // Resource is released here |

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

| **Principles(s):**  2: **Heed Compiler Warnings** - Compilers often provide warnings related to resource leaks. Addressing these warnings helps to ensure proper resource management.  8: **Practice Defense in Depth** - Proper resource management acts as a defense mechanism by ensuring that resources are released, preventing memory leaks and exhaustion.  9: **Use Effective Quality Assurance Techniques** - Regular code testing, including the use of static analysis tools, ensures that resources are managed efficiently and leaks are detected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | Resource Leak | Detects potential resource leaks, such as unclosed file handles. |
| Valgrind | 3.17.0 | Memcheck | Identifies memory leaks, improper memory usage, and resource mismanagement at runtime. |
| Coverity | 2024.1 | CERT.STR31-C | Checks for proper resource management, ensuring that resources like files are properly released. |
| Cppcheck | 2.8 | Resource Management | Analyzes code for potential resource leaks, such as missing resource deallocation. |
| AddressSanitizer | 12.0.1 | Resource Use | Identifies resource leaks and misuse during program execution. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Error Handling** | STD-010-CPP | Ensures errors are handled in a secure and consistent manner, preventing unintended behavior or leakage of sensitive information. |

| **Noncompliant Code** |
| --- |
| Errors are not handled, and the program continues in an inconsistent state, leading to crashes or unexpected behavior. |
| int divide(int a, int b) {  return a / b; // No check for division by zero  } |

| **Compliant Code** |
| --- |
| Errors are properly handled, ensuring the program behaves as expected even in exceptional cases. |
| int divide(int a, int b) {  if (b == 0) {  std::cerr << "Error: Division by zero" << std::endl;  return 0; // Handle error, avoid crash  }  return a / b;  } |

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

| **Principles(s):**  4: **Keep It Simple** - Proper error handling ensures the code remains simple and predictable, avoiding undefined behavior and making it easier to debug and maintain.  8: **Practice Defense in Depth** - Implementing error handling is an essential layer of defense, protecting the application from crashes or unexpected behavior that could lead to vulnerabilities.  9: **Use Effective Quality Assurance Techniques** - Testing error conditions and validating that the application handles errors securely is part of a robust quality assurance process. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | Error Handling | Identifies cases of missing or improper error handling, especially in critical parts of the code. |
| SonarQube | 9.3 | C | Detects instances where error conditions are not properly managed, suggesting safe error handling practices. |
| Coverity | 2024.1 | CERT.ERR08-C | Checks for comprehensive error handling, ensuring that all error cases are managed securely. |
| Cppcheck | 2.8 | Error Management | Analyzes code for potential flaws in error handling and suggests corrections to avoid crashes. |

### 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 will play a critical role in enforcing the standards outlined in this security policy. Within the existing DevOps process at Green Pace, the integration of security tools and practices will allow the organization to automate vulnerability scanning, code analysis, and policy compliance checks at every stage of the development pipeline.

According to the DevSecOps diagram, the automation should be embedded during the "Verify and Test" phase in pre-production and continue throughout production. This includes:

* Design Phase: Incorporating security testing as part of design requirements to ensure secure coding practices are integrated from the beginning.
* Build Phase: Utilizing static analysis tools like SonarQube and Clang Static Analyzer to identify code vulnerabilities and compliance issues during the build.
* Verify and Test Phase: Automating testing to check for memory leaks, input validation errors, and proper resource management before deployment.
* Monitor and Detect Phase: Using automated monitoring tools to continuously assess the production environment for compliance and detect potential security breaches or violations of the coding standards. By embedding automation into the DevSecOps pipeline, Green Pace ensures that coding standards are enforced, potential vulnerabilities are detected early, and compliance is maintained consistently.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | High | 1 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-006-CPP | Medium | Likely | Low | Medium | 2 |
| STD-007-CPP | High | Likely | Medium | High | 1 |
| STD-008-CPP | High | Likely | Medium | High | 1 |
| STD-009-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-010-CPP | High | Likely | Medium | High | 1 |

### 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 is the encryption of data when it is stored on disk. This policy ensures that sensitive information remains protected even if the storage is compromised. The policy applies to all customer data, internal records, and sensitive information stored in Green Pace's servers or databases.  **When the Policy Applies**: This policy applies during data storage and backup processes. Data must be encrypted using strong encryption algorithms before writing to storage devices. |
| Encryption in flight | **Explanation**: Encryption in flight protects data as it travels across networks. It uses protocols like TLS/SSL to secure data transmitted between clients, servers, and APIs. This prevents interception and eavesdropping by unauthorized entities.  **When the Policy Applies**: Applies during data transmission between internal systems, customer applications, and external networks. |
| Encryption in use | **Explanation**: Encryption in use secures data while it is actively being processed in memory. This can involve techniques like hardware-based memory encryption to protect sensitive data during processing.  **When the Policy Applies**: Applies when handling sensitive information in applications that require real-time processing, such as financial transactions or personal information analysis. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Explanation**: Authentication is the process of verifying the identity of users and systems. It ensures that only authorized individuals can access system resources, applications, and data. This policy requires the use of strong, multi-factor authentication methods for all users.  **When the Policy Applies**: Applies during all access attempts to systems, networks, applications, and data. |
| Authorization | **Explanation**: Authorization determines what authenticated users are allowed to do within the system. This policy ensures that users have access only to the resources and data necessary for their roles, adhering to the principle of least privilege.  **When the Policy Applies**: Applies during access control checks for resources, data, and application functionality. |
| Accounting | **Explanation**: Accounting involves recording user activities within the system to maintain an audit trail. This policy ensures that all access, modifications, and interactions with sensitive data are logged for review and analysis.  **When the Policy Applies**: Applies continuously, with logging and monitoring tools recording events such as login attempts, data access, and system modifications. |

**\***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 |  |
| 2.0 | 10/13/2024 | Project One Complete | Sean Glass |  |
|  |  |  |  |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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