**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc174227075)

[Contents 1](#_Toc174227076)

[Overview 3](#_Toc174227077)

[Purpose 3](#_Toc174227078)

[Scope 3](#_Toc174227079)

[Module Three Milestone 3](#_Toc174227080)

[Ten Core Security Principles 3](#_Toc174227081)

[C/C++ Ten Coding Standards 4](#_Toc174227082)

[Coding Standard 1 5](#_Toc174227083)

[Coding Standard 2 7](#_Toc174227084)

[Coding Standard 3 9](#_Toc174227085)

[Coding Standard 4 11](#_Toc174227086)

[Coding Standard 5 13](#_Toc174227087)

[Coding Standard 6 15](#_Toc174227088)

[Coding Standard 7 17](#_Toc174227089)

[Coding Standard 8 19](#_Toc174227090)

[Coding Standard 9 21](#_Toc174227091)

[Coding Standard 10 23](#_Toc174227092)

[Defense-in-Depth Illustration 25](#_Toc174227093)

[Project One 25](#_Toc174227094)

[Revise the C/C++ Standards 25](#_Toc174227095)

[Risk Assessment 25](#_Toc174227096)

[Automated Detection 25](#_Toc174227097)

[Automation 25](#_Toc174227098)

[Summary of Risk Assessments 27](#_Toc174227099)

[Create Policies for Encryption and Triple A 27](#_Toc174227100)

[Map the Principles 28](#_Toc174227101)

[Audit Controls and Management 30](#_Toc174227102)

[Enforcement 30](#_Toc174227103)

[Exceptions Process 30](#_Toc174227104)

[Distribution 31](#_Toc174227105)

[Policy Change Control 31](#_Toc174227106)

[Policy Version History 31](#_Toc174227107)

[Appendix A Lookups 31](#_Toc174227108)

[Approved C/C++ Language Acronyms 31](#_Toc174227109)

## 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 all input data is crucial to prevent common vulnerabilities such as SQL injection, buffer overflows, and cross-site scripting (XSS). This practice ensures that the data conforms to expected formats, ranges, and types, thus maintaining the integrity and security of the application. |
| 1. Heed Compiler Warnings | Compiler warnings often highlight potential vulnerabilities or unstable code. By addressing these warnings promptly, developers can identify and fix issues early, leading to more secure and reliable software. |
| 1. Architect and Design for Security Policies | Security should be an integral part of the software development lifecycle, embedded into the architecture and design phase. This ensures that security controls are considered from the outset, making the design resilient against potential attacks. |
| 1. Keep It Simple | Simple designs are easier to understand, implement, and audit, which inherently reduces the risk of security flaws. Complexity often introduces hidden vulnerabilities that attackers can exploit. |
| 1. Default Deny | This principle ensures that access is denied by default, and permissions are explicitly granted only where necessary. By doing so, the attack surface is minimized, and only authorized actions are permitted, enhancing overall security. |
| 1. Adhere to the Principle of Least Privilege | Limiting the access rights of users and processes to the minimum necessary to perform their functions reduces the potential impact of security breaches. This principle ensures that even if an account is compromised, the damage is contained. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before it is transmitted to other systems is crucial in maintaining data integrity. This process helps prevent the spread of malicious content and reduces the risk of injection attacks, ensuring that only clean and accurate data is shared. |
| 1. Practice Defense in Depth | Employing multiple layers of security controls creates a robust defense strategy. This ensures that even if one control fails, additional safeguards are in place to protect data and systems, significantly reducing the risk of successful attacks. |
| 1. Use Effective Quality Assurance Techniques | Regular testing, code reviews, and static analysis are key techniques for identifying and resolving security vulnerabilities. Ensuring that no unexpected behavior occurs is also critical, as it helps maintain a secure and stable codebase. |
| 1. Adopt a Secure Coding Standard | Adopting a recognized secure coding standard, such as the SEI CERT C++ Coding Standard, ensures that code adheres to industry best practices for security. This approach helps prevent common vulnerabilities and promotes consistency in coding practices, leading to more secure software development. |

### 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** | **Do not declare or define a reserved identifier** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Declaring or defining reserved identifiers can cause undefined behavior, conflicts with standard library names, or future compatibility issues. Reserved identifiers include names that start with an underscore followed by an uppercase letter or names that contain double underscores. |

| **Noncompliant Code** |
| --- |
| The code uses identifiers that start with an underscore followed by an uppercase letter and double underscores, which are reserved and can lead to undefined behavior or conflicts. |
| int \_MyVariable = 10; // Reserved identifier  double \_\_myFunction(double x) { // Reserved identifier  return x \* x;  } |

| **Compliant Code** |
| --- |
| The code uses identifiers that are not reserved, ensuring there are no conflicts or undefined behavior. |
| int myVariable = 10; // Non-reserved identifier  double myFunction(double x) { // Non-reserved identifier  return x \* x;  } |

| **Principles(s):**  (1) Validate Input Data – Using reserved identifiers can introduce risks since they might be interpreted in unexpected ways by the compiler, leading to undefined behavior. By avoiding these identifiers, you ensure that your code behaves as expected, which is a form of validating that your "input data" (in this case, the identifiers you use) is safe and correct.  (9) Use Effective Quality Assurance Techniques - Adhering to coding standards like avoiding reserved identifiers is a part of effective quality assurance. By ensuring that your identifiers are non-reserved, you're applying a preventive measure to maintain the quality and predictability of your code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | reserved-identifier | Automatic detection tool.  Partially checked |
| Clang | 3.9 | -Wreserved-id-macro  -Wuser-defined-literal | Compiler.  The -Wreserved-id-macro flag isn't enabled by default or with -Wall, but it is enabled with -Weverything. However, this flag doesn't catch all cases of this rule, like redefining reserved names. |
| Polyspace Bug Finder | R2024a | CERT C++: DCL51-CPP | Static analysis tool that automatically detects bugs, coding rule violations, security vulnerabilities, and other issues.  Checks for redefinitions of reserved identifiers (rule partially covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Converting between integer types can result in data loss or misinterpretation if the target type cannot represent the value of the original type. Ensuring safe conversions prevents unexpected behavior and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The code converts a large integer value to a smaller integer type without checking if the conversion will result in data loss. |
| void convert(int largeValue) {  short smallValue = largeValue; // Potential data loss  std::cout << smallValue << std::endl;  } |

| **Compliant Code** |
| --- |
| The code checks if the large integer value is within the range of the smaller integer type before performing the conversion, preventing data loss. |
| void convert(int largeValue) {  if (largeValue > std::numeric\_limits<short>::max() || largeValue < std::numeric\_limits<short>::min()) {  throw std::out\_of\_range("Value out of range for short");  }  short smallValue = static\_cast<short>(largeValue); // Safe conversion  std::cout << smallValue << std::endl;  } |

| **Principles(s):**  (1) Validate Input Data – This principle is about ensuring that the data processed by a program is within expected and acceptable ranges. This is crucial for preventing errors, vulnerabilities, and undefined behavior in the software. By not validating the input data before conversion, the code risks introducing bugs and vulnerabilities related to data integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | memsetValueOutOfRange | Static Analysis tool.  The second argument to memset() cannot be represented as unsigned char |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space of character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensuring that storage for strings includes space for the null terminator prevents buffer overflows and data corruption. This practice is essential for maintaining the integrity and correctness of string operations. |

| **Noncompliant Code** |
| --- |
| The code copies a string without ensuring that the destination buffer has enough space for the source string and the null terminator, leading to potential buffer overflow. |
| void copyString(char\* dest, const char\* src) {  strcpy(dest, src); // No check for sufficient space  } |

| **Compliant Code** |
| --- |
| The code checks if the destination buffer has enough space for the source string and the null terminator before copying, preventing buffer overflow. |
| void copyString(char\* dest, const char\* src, size\_t destSize) {  if (strlen(src) + 1 > destSize) {  throw std::overflow\_error("Destination buffer is too small");  }  strncpy(dest, src, destSize - 1);  dest[destSize - 1] = '\0'; // Ensure null termination  } |

| **Principles(s):**  (7) Sanitize Data Sent to Other Systems – This principle is mapped to the need for careful validation and checking of buffer sizes before performing string operations. The compliant code example follows this principle by ensuring that the destination buffer is sufficiently large to hold the string and its null terminator, thereby preventing buffer overflows and ensuring safe data handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM | Static Analysis tool.  Buffer overrun  Type overrun  No space for null terminator |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Static Analysis tool.  Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Including user input directly in format strings can lead to vulnerabilities such as SQL injection. Ensuring user input is excluded from format strings helps prevent these attacks by ensuring input is properly sanitized and parameterized. |

| **Noncompliant Code** |
| --- |
| The code constructs an SQL query using user input directly, making it vulnerable to SQL injection. |
| std::string query = "SELECT \* FROM users WHERE id = " + userInput;  db.execute(query); // Vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| In this compliant solution, the code uses a prepared statement to safely include user input in the SQL query, preventing SQL injection. |
| std::string query = "SELECT \* FROM users WHERE id = ?";  db.execute(query, userInput); // Use parameterized queries |

| **Principles(s):**  (1) Validate Input Data – This principle is crucial in preventing the introduction of vulnerabilities, such as SQL injection, by ensuring that input meets the necessary criteria before being processed.  (7) Sanitize Data Sent to Other Systems – This principle involves ensuring that any data passed from your system to another is clean, safe, and free from elements that could cause harm, such as malicious code or improperly formatted data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | IO.INJ.FMT  MISC.FMT | Static Analysis tool.  Format string injection  Format string |
| GCC | 4.3.4 |  | Compiler system that with built-in static analysis capabilities.  Can detect violations of this rule when the -Wformat-security flag is used |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Accessing freed memory can lead to undefined behavior, crashes, and security vulnerabilities. Ensuring that memory is not accessed after it has been freed is essential for maintaining the stability and security of an application. |

| **Noncompliant Code** |
| --- |
| The code accesses memory that has been freed, leading to undefined behavior. |
| void useFreedMemory() {  int\* ptr = new int[10];  delete[] ptr;  // Accessing freed memory  int value = ptr[0];  std::cout << value << std::endl;  } |

| **Compliant Code** |
| --- |
| The code sets the pointer to nullptr after freeing the memory to prevent accessing freed memory. |
| void useFreedMemory() {  int\* ptr = new int[10];  delete[] ptr;  ptr = nullptr; // Avoid accessing freed memory  // Safe access  if (ptr != nullptr) {  int value = ptr[0];  std::cout << value << std::endl;  }  } |

| **Principles(s):**  (8) Practice Defense in Depth – This principle is mapped to the need for multiple layers of protection against accessing freed memory. The compliant code example adheres to this principle by implementing both memory management and additional checks to prevent unsafe access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.UAF | Static Analysis tool.  The ALLOC.UAF checker specifically detects instances of use-after-free (UAF) vulnerabilities, where a program continues to access memory after it has been freed. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Static assertions validate assumptions at compile time, ensuring that certain conditions are met before the program runs. This helps catch errors early and ensures the correctness of constant expressions. |

| **Noncompliant Code** |
| --- |
| The code does not check at compile time if bufferSize exceeds maxSize, which could lead to potential issues if the buffer size is incorrectly set. |
| const int bufferSize = 1024;  const int maxSize = 512;  // Noncompliant: No compile-time check for buffer size  char buffer[bufferSize]; |

| **Compliant Code** |
| --- |
| The code uses a static assertion to check at compile time if bufferSize exceeds maxSize, preventing potential issues by ensuring the condition is met. |
| const int bufferSize = 1024;  const int maxSize = 512;  // Compliant: Static assertion ensures buffer size is within the limit  static\_assert(bufferSize <= maxSize, "Buffer size exceeds maximum allowed size");  char buffer[bufferSize]; |

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques - The compliant code adheres to the principle of using effective quality assurance techniques by incorporating a static\_assert to validate the buffer size at compile time. The use of static\_assert demonstrates a proactive approach to quality assurance, aligning with the principle of catching errors early and maintaining high code quality. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Compiler with static analysis capabilities.  Checked by clang-tidy  The misc-static-assert checker in Clang is used to detect improper usage or the absence of static\_assert statements where they should be applied. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Guarantee exception safety** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exception safety ensures that a program remains in a valid state and that resources are properly managed when exceptions occur. This practice prevents resource leaks and inconsistent states, maintaining the reliability and stability of the application. |

| **Noncompliant Code** |
| --- |
| The code allocates memory and calls a function that might throw an exception. If an exception is thrown, the memory allocated to ptr might not be freed, leading to a memory leak. |
| void process() {  int\* ptr = new int[10];  // Potential exception thrown here  someFunctionThatMightThrow();  delete[] ptr; // This might not be executed if an exception is thrown  } |

| **Compliant Code** |
| --- |
| The code uses a smart pointer (std::unique\_ptr) to manage the allocated memory. This ensures that the memory is automatically freed when the pointer goes out of scope, even if an exception is thrown. |
| void process() {  std::unique\_ptr<int[]> ptr(new int[10]); // Use smart pointer for automatic resource management  someFunctionThatMightThrow();  // Memory is automatically freed when ptr goes out of scope  } |

| **Principles(s):**  (8) Practice Defense in Depth – The compliant code adds an additional layer of protection by using smart pointers. Even if the function throws an exception, the smart pointer ensures that the memory is still freed, providing a secondary defense against resource leaks.  (9) Use Effective Quality Assurance Techniques – By using smart pointers, the compliant code implements a robust resource management strategy that ensures memory is managed correctly regardless of whether an exception is thrown. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Leak |
| ParaSoft C/C++test | 2023.1 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Static analysis and testing tool for C and C++ code. It supports compliance with various coding standards, including CERT C++ rules.  Always catches exceptions  Do not leave ‘catch’ blocks empty |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input Output | STD-008-CPP | Properly closing files when they are no longer needed ensures that resources are released, preventing resource leaks and maintaining application stability. This practice is crucial for managing system resources effectively and avoiding file descriptor exhaustion. |

| **Noncompliant Code** |
| --- |
| The code opens a file but does not ensure that the file is closed if an exception occurs, leading to potential resource leaks. |
| void processFile(const std::string& filename) {  std::fstream file;  file.open(filename, std::ios::in);  if (!file.is\_open()) {  throw std::runtime\_error("Failed to open file");  }  // Read from the file  // File is not closed if an exception occurs  } |

| **Compliant Code** |
| --- |
| The code ensures that the file is closed after processing or if an exception occurs, preventing resource leaks and maintaining application stability. |
| void processFile(const std::string& filename) {  std::fstream file;  file.open(filename, std::ios::in);  if (!file.is\_open()) {  throw std::runtime\_error("Failed to open file");  }  try {  // Read from the file  } catch (...) {  file.close(); // Ensure file is closed in case of an exception  throw; // Rethrow the exception  }  file.close(); // Ensure file is closed after processing  } |

| **Principles(s):**  (8) Practice Defense in Depth – This principle is mapped because of the use of exception handling and explicit file closing in the compliant code example. By incorporating a try-catch block and ensuring that the file is closed regardless of whether an exception occurs, the code provides multiple layers of defense against resource leaks and instability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-FIO51-a | Checks for resource leak (rule partially covered)  The CERT\_CPP-FIO51-a checker ensures that files are properly closed after they are no longer needed, preventing resource leaks and ensuring application stability by enforcing best practices in file handling. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| Memory Management | STD-009-CPP | Properly detecting and handling memory allocation errors ensures that a program can gracefully handle out-of-memory conditions and prevent undefined behavior. This practice helps maintain the stability and reliability of the application. |

| **Noncompliant Code** |
| --- |
| The code allocates memory without handling potential allocation failures, leading to possible undefined behavior if an exception is thrown. |
| void allocateMemory() {  int\* ptr = new int[1000]; // Potentially throws std::bad\_alloc  // No check for allocation failure  // Use ptr  delete[] ptr;  } |

| **Compliant Code** |
| --- |
| The code uses a try-catch block to handle potential memory allocation failures, ensuring that the program can respond appropriately to out-of-memory conditions. |
| void allocateMemory() {  try {  int\* ptr = new int[1000];  // Use ptr  delete[] ptr;  } catch (const std::bad\_alloc& e) {  std::cerr << "Memory allocation failed: " << e.what() << std::endl;  // Handle error appropriately  }  } |

| **Principles(s):**  (8) Practice Defense in Depth – The compliant code adheres to the principle of Defense in Depth by adding a secondary layer of protection through the try-catch block. If the primary operation (memory allocation) fails, the catch block provides a fallback mechanism to handle the failure, preventing the program from crashing or entering an undefined state.  (9) Use Effective Quality Assurance Techniques - By including a try-catch block to handle potential memory allocation failures, the compliant code implements effective quality assurance techniques that ensure the program's stability and reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled.  The CHECKED\_RETURN checker ensures that functions that return error codes or important values are properly handled by the caller. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not return from a function declared [[noreturn]]** |
| --- | --- | --- |
| Miscellaneous | STD-010-CPP | Functions declared with [[noreturn]] are expected not to return to the caller. Returning from such functions can lead to undefined behavior, as the compiler and runtime may make optimizations based on the assumption that the function will not return. |

| **Noncompliant Code** |
| --- |
| The code returns from a function declared with [[noreturn]], which can lead to undefined behavior. |
| [[noreturn]] void fatalError() {  // Perform some error handling  return; // Incorrect: returning from a [[noreturn]] function  } |

| **Compliant Code** |
| --- |
| The code properly handles the error by calling std::exit, ensuring that the function does not return. |
| [[noreturn]] void fatalError() {  // Perform some error handling  std::exit(EXIT\_FAILURE); // Correct: exit or throw an exception  } |

| **Principles(s):**  (8) Practice Defense in Depth – The compliant code follows the principle of Defense in Depth by ensuring that the [[noreturn]] function cannot return. The compliant also declares the fatalError() function with the [[noreturn]] attribute, but instead of returning, it calls std::exit(EXIT\_FAILURE); to terminate the program.  (9) Use Effective Quality Assurance Techniques – By ensuring that the [[noreturn]] function does not return, the code avoids undefined behavior and maintains the stability and reliability of the application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Winvalid-noreturn | The -Winvalid-noreturn checker issues warnings when a function marked with [[noreturn]] returns a value or completes execution without terminating the program.­­­­­ |
| SonarQube C/C++ Plugin | 4.10 | S935 | Static code analysis tool that identifies bugs, vulnerabilities, and code smells in C and C++ codebases. The S935 checker ensures that functions marked with [[noreturn]] do not return, thereby preventing potential undefined behavior and improving code reliability. |

### 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 crucial to ensuring that the security standards defined in this policy are consistently enforced and adhered to throughout the software development and operational lifecycle. By integrating DevSecOps principles, Green Pace can improve system security without sacrificing the speed and flexibility that DevOps provides.

In the pre-production phase, the focus will be on integrating security early in the SDLC. During the planning stage, threat modeling, impact analysis, and prioritizing security tasks based on potential risks should be automated. Tools such as threat modeling frameworks can be incorporated to automatically identify and assess risks in the design phase. Incorporate automated security checks during the design phase to ensure best practices are followed, such as using DevSecOps principles like OWASP Top 10 as references. Additionally, configure static code analysis tools to run automatically on code repositories to detect vulnerabilities early in the development process. Next, add continuous security testing to the CI/CD pipeline. This can include automated unit tests, static analysis, and security checks that run with each build. This ensures that code is regularly checked against established security standards, like input validation and memory management. Finally, automated vulnerability scanning and penetration testing tools should be employed to continuously evaluate the codebase for security weaknesses. Tools like cppcheck for C++ can be used to ensure compliance with secure coding standards, and automated testing frameworks can be configured to validate security requirements.

In the production phase, automation ensures that security is maintained throughout the system's operation. Green Pace should automate the deployment of security settings and continuous monitoring tools, including health checks that verify application integrity after deployment. Use automated tools for real-time monitoring, detecting unusual activities, and responding to security incidents. Implement automated security patching and updates to keep systems secure, with rollback or alert systems in place for any issues. Set up automated incident response protocols to block threats, isolate networks, and perform post-incident analysis using predefined scripts and playbooks.

### 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 | Low | Unlikely | Low | Low | 3 |
| STD-002-CPP | High | Probable | High | Medium | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | High | Low | 3 |
| STD-007-CPP | High | Likely | High | Medium | 2 |
| STD-008-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-009-CPP | High | Likely | Medium | High | 1 |
| STD-010-CPP | Medium | Unlikely | Low | Low | 3 |

### 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 | Encryption at rest means protecting data stored on disk or any long-term storage. It ensures that data is encrypted when not in use, preventing unauthorized access if the storage is compromised.  This policy covers all data stored in the organization’s databases, file systems, and backups. It recommends using encryption algorithms like AES-256 to secure data at rest.  Encrypting data at rest is crucial to protect sensitive information from breaches, especially in scenarios where physical access to storage devices is a concern.  Encrypting data at rest is crucial to protect sensitive information from breaches, especially in scenarios where physical access to storage devices is a concern. |
| Encryption in flight | Encryption in flight means protecting data while it's being transmitted over networks. It ensures that data is encrypted during transit, preventing unauthorized interception.  This policy applies to all data sent over the internet, internal networks, or any communication channels. Encryption protocols like TLS (Transport Layer Security) are commonly used to secure data during transit.  Encrypting data in transit is essential to safeguard sensitive information from being intercepted or altered during transmission, particularly over public networks. |
| Encryption in use | Encryption in use means protecting data that is actively being processed or accessed in memory. It ensures that data stays secure even while it's being used by applications or services.  This policy applies to situations where data is being processed in memory, like during computations or when it's loaded into an application. Techniques such as homomorphic encryption or secure enclaves can be used to protect the data.  Encrypting data in use protects sensitive information from being exposed during processing, especially in environments where multiple users or services may have access to shared resources. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of confirming the identity of users or systems before allowing access to resources. This is usually done with credentials like usernames and passwords, biometrics, or multi-factor authentication (MFA).  This policy applies to all user logins, system access, and interfaces that require identity verification. Strong password policies and the use of MFA are recommended practices.  Authentication ensures that only authorized individuals can access sensitive data and systems, reducing the risk of unauthorized access and potential breaches. |
| Authorization | Authorization defines what an authenticated user or system can do after access is granted. It involves setting permissions and access levels for different roles within the organization.  This policy applies to all systems and applications that handle user permissions and access controls. Role-based access control (RBAC) is a common method used to enforce these authorization policies.  Authorization ensures that users only have access to the resources necessary for their role, thereby minimizing the risk of data leakage and unauthorized actions. |
| Accounting | Accounting involves tracking and recording user activities and system events. This helps monitor access and changes to data, creating an audit trail for security incidents.  This policy applies to all systems and applications where logging and monitoring user actions are needed. Using detailed logging systems and conducting regular audits are important practices.  Accounting makes sure that all user actions are recorded, allowing the organization to detect and respond to suspicious activities and maintain accountability. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**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 | Rajiv Dialani |
| 2.0 | 07/21/2024 | Ten Core Security Principle  Part of C/C++ Ten Coding Standards | Rajiv Dialani |  |
| 3.0 | 08/10/2024 | Edited Ten Core SP  Edited C/C++ Coding Standards  Finished Security Policy | Rajiv Dialani |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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