**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 | Always validate all input data from untrusted sources before processing. This prevents malicious data from causing unintended behavior, such as injection attacks or buffer overflows, ensuring data integrity and application security. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential issues in the code that could lead to security vulnerabilities. Developers should treat warnings as errors and resolve them to prevent undefined behaviors from reaching production code. |
| 1. Architect and Design for Security Policies | Integrate security considerations into the architecture and design phases. By planning for security from the outset, systems can be built with robust defenses against potential attacks, reducing vulnerabilities inherent in the system's design. |
| 1. Keep It Simple | |  | | --- | | Simplify code and system designs to reduce complexity, which is a common source of security flaws. Simple designs are easier to understand, test, and secure, minimizing the risk of introducing errors that could be exploited. | |
| 1. Default Deny | Implement a security posture where, by default, access is denied, and permissions are explicitly granted as needed. This limits exposure by ensuring that only authorized users and processes can access system resources. |
| 1. Adhere to the Principle of Least Privilege | Limit users and processes to the minimal level of access required to perform their functions. This reduces the potential impact of accidental or malicious misuse of privileges, enhancing overall system security. |
| 1. Sanitize Data Sent to Other Systems | Cleanse all data before sending it to other systems to prevent the propagation of malicious content. This protects downstream systems from attacks like injection or cross-site scripting by ensuring data is safe to process. |
| 1. Practice Defense in Depth | Employ multiple layers of security controls throughout the system. If one layer fails, additional layers provide continued protection, making it more difficult for attackers to compromise the system. |
| 1. Use Effective Quality Assurance Techniques | Implement thorough testing, code reviews, and static analysis to identify and remediate security vulnerabilities early in the development lifecycle, improving code quality and security. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards to ensure consistency and the application of best practices. This reduces the likelihood of introducing security flaws due to coding errors or misunderstandings. |

### 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 | Avoid Implicit Type Conversions  - Implicit type conversions can lead to unexpected behavior, data loss, or security vulnerabilities. By avoiding implicit conversions, we ensure data integrity and prevent subtle bugs. |

| **Noncompliant Code** |
| --- |
| Implicit conversion from a larger type to a smaller type can cause data loss. |
| unsigned long largeValue = 5000000000;  unsigned int smallValue = largeValue; // Implicit conversion may truncate data |

| **Compliant Code** |
| --- |
| Explicitly handle type conversion to ensure data integrity. |
| unsigned long largeValue = 5000000000;  if (largeValue <= UINT\_MAX) {  unsigned int smallValue = static\_cast<unsigned int>(largeValue);  // Proceed with smallValue  } else {  // Handle the error condition  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   1. **Validate Input Data:** Prevents unintended data conversions that might corrupt input data integrity or lead to errors. 2. **Heed Compiler Warnings:** Ensures developers address warnings related to type mismatches during compilation. 3. **Use Effective Quality Assurance Techniques:** Encourages thorough testing and review to identify unintended type conversions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S1871 | Detects implicit type conversions in C++ code to ensure strong type safety. |
| Clang-Tidy | 15.0 | readability-implicit-cast | Flags instances of implicit type conversions, ensuring code clarity and safety. |
| Coverity | 2023.12 | Type Mismatch Checker | Identifies potential mismatches and implicit conversions that could lead to errors |
| Cppcheck | 2.10 | typeCastCheck | Highlights implicit or unsafe typecasting in C++ projects. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Check Return Values of Functions  - Ignoring return values of functions, especially those that report errors, can lead to unexpected behavior and security vulnerabilities. Always check return values to handle errors appropriately. |

| **Noncompliant Code** |
| --- |
| Fails to check the return value of fgets, which may result in using uninitialized data. |
| char buffer[256];  fgets(buffer, sizeof(buffer), stdin);  // Assumes buffer contains valid data  processInput(buffer); |

| **Compliant Code** |
| --- |
| Checks the return value of fgets before using the data. |
| char buffer[256];  if (fgets(buffer, sizeof(buffer), stdin) != NULL) {  processInput(buffer);  } else {  // Handle error condition  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Validate Input Data:** Ensures the validity of data before processing by checking function return values. * **Heed Compiler Warnings:** Helps developers catch unhandled return values, which compilers may warn about. * **Use Effective Quality Assurance Techniques:** Testing and static analysis tools ensure return values are correctly handled. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S3626 | Detects unhandled return values in function calls. |
| Clang-Tidy | 15.0 | misc-ignored-return-value | Flags calls to functions whose return values are ignored. |
| Coverity | 2023.12 | Return Value Check | Identifies unhandled function return values that may lead to security issues. |
| Cppcheck | 2.10 | checkReturn | Highlights ignored return values for critical functions in C/C++ codebases. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP | Properly Null-Terminate Strings  - Failure to null-terminate strings can result in buffer overflows and security vulnerabilities. Ensure all strings are properly null-terminated to prevent reading or writing beyond the intended memory. |

| **Noncompliant Code** |
| --- |
| Uses strncpy without ensuring null termination, which may result in non-null-terminated strings. |
| char dest[10];  strncpy(dest, src, sizeof(dest));  // dest may not be null-terminated |

| **Compliant Code** |
| --- |
| Ensures the destination string is null-terminated. |
| char dest[10];  strncpy(dest, src, sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Validate Input Data:** Ensures string data is correctly formatted and terminated to prevent buffer overflows and unexpected behavior. * **Keep It Simple:** Properly null-terminating strings simplifies memory management and reduces errors. * **Practice Defense in Depth:** Adds a safeguard against vulnerabilities caused by improper memory access. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S5537 | Identifies cases where strings may not be null-terminated. |
| Clang-Tidy | 15.0 | bugprone-string-constructor | Flags improper string handling and termination issues. |
| Coverity | 2023.12 | Buffer Overrun Checker | Detects improper memory handling, including string null-termination issues. |
| Cppcheck | 2.10 | nullTermination | Identifies strings that may not be null-terminated, preventing memory errors. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Use Prepared Statements for Database Queries  - Using prepared statements with parameterized queries helps prevent SQL injection attacks by separating SQL code from data. |

| **Noncompliant Code** |
| --- |
| Concatenates user input directly into SQL query, making it vulnerable to SQL injection. |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  executeQuery(query); |

| **Compliant Code** |
| --- |
| Uses a prepared statement to safely include user input. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement\* stmt = conn->prepareStatement(query);  stmt->setString(1, username);  ResultSet\* rs = stmt->executeQuery(); |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Validate Input Data:** Prepared statements ensure that user inputs are treated as data, not executable code, preventing injection attacks. * **Default Deny:** Restricts SQL execution to explicitly defined queries and parameters, reducing exposure to unauthorized actions. * **Practice Defense in Depth:** Adds a layer of security against injection attacks, complementing other access controls. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S3649 | Detects SQL queries that are vulnerable to injection due to string concatenation. |
| Clang-Tidy | 15.0 | bugprone-sql-injection | Flags SQL queries constructed using unsafe string concatenation. |
| Coverity | 2023.12 | SQL Injection Checker | Identifies potential SQL injection vulnerabilities in code. |
| OWASP ZAP | 2.13.0 | SQL Injection Scanner | Scans web applications for SQL injection vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Free Dynamically Allocated Memory  - Failing to release dynamically allocated memory leads to memory leaks, which can exhaust system resources and lead to application failure or denial of service. |

| **Noncompliant Code** |
| --- |
| Allocates memory but never releases it. |
| int\* data = new int[100];  // ... use data  // Memory is not freed |

| **Compliant Code** |
| --- |
| Releases allocated memory when it is no longer needed. |
| int\* data = new int[100];  // ... use data  delete[] data;  data = nullptr; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Practice Defense in Depth:** Ensures resource management and protection against memory-related vulnerabilities like exhaustion and leaks. * **Use Effective Quality Assurance Techniques:** Employs tools and testing methods to identify memory leaks during the development lifecycle. * **Adopt a Secure Coding Standard:** Adhering to secure memory management practices ensures consistency and reduces the risk of vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | Memcheck | Detects memory leaks and improper memory use in C++ programs. |
| SonarQube | 9.6 | cpp:S1751 | Identifies cases of unreleased dynamically allocated memory. |
| Clang-Tidy | 15.0 | cert-mem54-cpp | Checks for memory leaks and proper cleanup of dynamically allocated resources. |
| Coverity | 2023.12 | Resource Leak Checker | Flags dynamically allocated memory that is not released. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Do Not Use Assertions for Error Handling  - Assertions are meant for detecting programming errors during development, not for handling runtime errors in production code. Using assertions for error handling may cause unexpected termination. |

| **Noncompliant Code** |
| --- |
| Uses assert to check for runtime errors, which can be disabled and may not handle errors appropriately. |
| int result = doSomething();  assert(result != -1);  // Proceed assuming success |

| **Compliant Code** |
| --- |
| Checks for errors and handles them appropriately. |
| int result = doSomething();  if (result == -1) {  // Handle error  } else {  // Proceed with result  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Heed Compiler Warnings:** Encourages addressing potential runtime errors rather than relying on assertions, which may not execute in production. * **Keep It Simple:** Ensures clear and maintainable error handling logic in the code, reducing the chance of unexpected behavior. * **Use Effective Quality Assurance Techniques:** Focuses on robust error checking during development rather than relying on runtime assertions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-err34-cpp | Detects misuse of assertions for error handling in C++ code. |
| SonarQube | 9.6 | cpp:S3603 | Identifies assertion misuse and recommends proper error handling methods. |
| Cppcheck | 2.10 | assertCheck | Flags inappropriate use of assertions for runtime error handling. |
| Coverity | 2023.12 | Assertion Checker | Highlights improper use of assertions that may fail in production environments. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Catch Exceptions by Reference  - Catching exceptions by value can lead to slicing and inefficient copies. Catching by reference ensures that the complete exception object is caught and handled. |

| **Noncompliant Code** |
| --- |
| Catches exceptions by value, which may result in object slicing. |
| try {  // Code that may throw  } catch (MyException e) {  // Handle exception  } |

| **Compliant Code** |
| --- |
| Catches exceptions by const reference to avoid slicing. |
| try {  // Code that may throw  } catch (const MyException& e) {  // Handle exception  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Keep It Simple:** Simplifies exception handling by avoiding the complexities of object slicing and ensures clarity in code. * **Use Effective Quality Assurance Techniques:** Promotes practices that prevent subtle bugs caused by inefficient or incomplete exception handling. * **Adopt a Secure Coding Standard:** Encourages consistent and efficient exception handling across all C++ projects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S1181 | Flags exception handling by value and recommends catching by reference. |
| Clang-Tidy | 15.0 | modernize-avoid-catch-by-value | Detects instances of catching exceptions by value and suggests using references. |
| Cppcheck | 2.10 | exceptionCheck | Highlights inefficient or unsafe exception handling practices. |
| Coverity | 2023.12 | Exception Handling Checker | Identifies improper exception handling, such as catching by value. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-008-CPP | Release Resources in All Paths  - Failing to release resources like file handles or network connections can lead to resource exhaustion. Ensure that resources are released in all execution paths, including exceptions. |

| **Noncompliant Code** |
| --- |
| Does not release file handle if an exception is thrown. |
| std::ifstream file("data.txt");  // Perform operations that may throw exceptions  processFile(file);  // File is not closed if an exception occurs |

| **Compliant Code** |
| --- |
| Uses RAII (Resource Acquisition Is Initialization) to ensure the file is closed even if an exception is thrown. |
| void processFile(const std::string& filename) {  std::ifstream file(filename);  if (!file) {  // Handle error  }  // File will be closed automatically when going out of scope  process(file);  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Practice Defense in Depth:** Ensures robust resource management, preventing resource exhaustion that could be exploited. * **Keep It Simple:** Simplifies code by automating resource release using RAII (Resource Acquisition Is Initialization). * **Adopt a Secure Coding Standard:** Promotes consistent practices to handle resources, reducing the risk of leaks and ensuring system stability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | MemCheck | Detects resource leaks, such as unclosed file handles and memory leaks. |
| SonarQube | 9.6 | cpp:S2095 | Flags unclosed resources like file handles or sockets. |
| Clang-Tidy | 15.0 | cert-err52-cpp | Identifies missing cleanup for dynamically or manually managed resources. |
| Coverity | 2023.12 | Resource Leak Checker | Identifies potential resource leaks in all execution paths, including exceptions. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-009-CPP | Do Not Ignore Exceptions  - Ignoring exceptions can lead to undefined behavior and security vulnerabilities. All exceptions should be appropriately handled to maintain application stability. |

| **Noncompliant Code** |
| --- |
| Catches exceptions but does nothing, hiding errors. |
| try {  performCriticalOperation();  } catch (const std::exception&) {  // Silently ignore the exception  } |

| **Compliant Code** |
| --- |
| Handles the exception or rethrows it after logging. |
| try {  performCriticalOperation();  } catch (const std::exception& e) {  // Handle exception appropriately  logError(e.what());  throw; // Rethrow exception if it cannot be handled here  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Use Effective Quality Assurance Techniques:** Ensures exceptions are logged and analyzed to identify and resolve issues. * **Keep It Simple:** Encourages straightforward and effective error handling rather than silent failures. * **Practice Defense in Depth:** Proper exception handling adds a layer of defense against unexpected errors that could compromise application stability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S2737 | Detects empty exception handlers that fail to process or log errors. |
| Clang-Tidy | 15.0 | bugprone-empty-catch | Flags catch blocks that do not properly handle or log exceptions. |
| Cppcheck | 2.10 | exceptionHandling | Identifies exception handlers that silently ignore exceptions without action. |
| Coverity | 2023.12 | Exception Handling Checker | Highlights cases where exceptions are caught but not processed effectively. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | STD-010-CPP | Avoid Data Races in Multithreaded Code  - Data races occur when multiple threads access shared data without proper synchronization, leading to undefined behavior. Use synchronization mechanisms to prevent data races. |

| **Noncompliant Code** |
| --- |
| Accesses shared data without synchronization. |
| int sharedCounter = 0;  void incrementCounter() {  for (int i = 0; i < 1000; ++i) {  ++sharedCounter; // Data race  }  } |

| **Compliant Code** |
| --- |
| Uses mutexes to synchronize access to shared data. |
| #include <mutex>  int sharedCounter = 0;  std::mutex counterMutex;  void incrementCounter() {  for (int i = 0; i < 1000; ++i) {  std::lock\_guard<std::mutex> lock(counterMutex);  ++sharedCounter; // Safe access  }  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.]   * **Practice Defense in Depth:** Ensures multiple layers of protection by using synchronization mechanisms to handle shared data safely. * **Use Effective Quality Assurance Techniques:** Encourages thorough testing and analysis to identify and prevent concurrency issues. * **Adopt a Secure Coding Standard:** Promotes consistent and effective use of thread-safe programming practices to avoid data races. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | Integrated into GCC/Clang | Data Race Detector | Dynamically detects data races in multithreaded applications during runtime. |
| Valgrind | 3.21.0 | Helgrind | Identifies data races and synchronization issues in C++ programs. |
| Clang-Tidy | 15.0 | cert-msc01-cpp | Flags shared variables accessed without proper synchronization mechanisms. |
| Coverity | 2023.12 | Concurrency Checker | Identifies data races and improper thread synchronization. |

### 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.

[Insert your written explanations here.]

### Automation Integration into the DevSecOps Pipeline

Automation is a critical component of enforcing and ensuring compliance with the standards outlined in this security policy. By embedding security checks and processes into the DevSecOps pipeline, Green Pace can achieve a robust, scalable, and continuous security posture. The following outlines the integration of automation at various stages of the development lifecycle, leveraging the Defense-in-Depth and DevSecOps diagrams:

**Plan: Address Technical Security Debt**

* During the planning phase, automation tools will be used to identify potential security vulnerabilities early.
* **Static Analysis Tools**: Tools like SonarQube or Cppcheck will scan the codebase for violations of secure coding standards, such as memory leaks, improper exception handling, and SQL injection vulnerabilities.
* **Threat Modeling**: Automated tools such as Microsoft Threat Modeling Tool will help assess potential attack vectors based on the system's architecture.

**Design: Build Secure Architectures**

* **Architecture Validation**: Automation tools will enforce secure design patterns and validate compliance with predefined secure coding principles, ensuring adherence to the principle of least privilege and defense in depth.
* These validations will be automated within CI/CD pipelines to ensure designs comply with Green Pace’s security standards.

**Build: Enforce Secure Compilation**

* During the build phase, automated dependency checkers like **OWASP Dependency-Check** will ensure that libraries and frameworks used are free from known vulnerabilities.
* **Compiler Settings**: Enable strict compiler warnings to treat potential coding issues as errors, forcing developers to address them before moving forward.

**Verify and Test: Comprehensive Security Testing**

* **Static Application Security Testing (SAST)**: Automated SAST tools will be integrated into the CI/CD pipeline to check for violations of secure coding standards, such as proper resource handling and null-termination of strings.
* **Dynamic Application Security Testing (DAST)**: Tools like OWASP ZAP will perform runtime testing of the application for vulnerabilities like unhandled exceptions and improper synchronization in multithreaded code.
* **Fuzz Testing**: Tools like AFL (American Fuzzy Lop) will simulate unexpected inputs to ensure robust error handling.

**Pre-Production: Transition and Health Check**

* **Integration Testing**: Automated integration testing tools will validate that all components, including third-party libraries, work securely together.
* **Software Signing**: Tools such as GPG will validate the integrity of builds before deployment to prevent tampering.

**Production: Monitor and Detect**

* **Runtime Application Self-Protection (RASP)**: Automation will continuously monitor the application in real-time for potential exploits, such as data races or improper exception handling.
* **Network Monitoring**: Tools like Splunk and Nagios will automatically alert on suspicious activity in the application or network infrastructure.

**Respond: Incident Response Automation**

* **SOAR Tools**: Security Orchestration, Automation, and Response (SOAR) platforms will streamline responses to identified vulnerabilities or breaches, ensuring incidents are addressed promptly.

**Adapt and Maintain**

* Regular automated scans using tools such as Valgrind and ThreadSanitizer will ensure ongoing compliance with secure coding practices.
* Configuration management tools like Ansible and Chef will maintain a consistent, secure environment.

By embedding automation into each phase of the DevSecOps pipeline, Green Pace ensures that secure coding standards are consistently enforced, reducing vulnerabilities and maintaining compliance with its defense-in-depth strategy.

### 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 | Low | High | 5 |
| STD-003-CPP | High | Possible | Medium | High | 4 |
| STD-004-CPP | Critical | Likely | Low | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 5 |
| STD-006-CPP | High | Likely | Medium | High | 4 |
| STD-007-CPP | Medium | Likely | Low | Medium | 3 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | High | Likely | Low | High | 5 |
| STD-010-CPP | Critical | Likely | Medium | High | 5 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | * **Definition:** Protects stored data by encrypting it using secure algorithms. * **Application:** Applied to databases, backups, and file systems storing sensitive data. * **Purpose:** Prevent unauthorized access to data in case of physical theft or system compromise. |
| Encryption in flight | * **Definition:** Secures data transmitted across networks. * **Application:** Used for securing HTTP (HTTPS), emails (TLS), and API communications. * **Purpose:** Prevents interception of sensitive data during transmission (e.g., man-in-the-middle attacks). |
| Encryption in use | * **Definition:** Encrypts data while being processed in memory. * **Application:** Applied to applications handling sensitive data like cryptographic keys or personal data. * **Purpose:** Prevents data leakage through memory dumps or unauthorized system access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | * **Definition:** Verifies the identity of users accessing the system. * **Application:** Implement strong multi-factor authentication for user logins and privileged operations. * **Purpose:** Prevent unauthorized access to critical resources. |
| Authorization | * **Definition:** Grants or denies user actions based on roles or policies. * **Application:** Use role-based access control (RBAC) to restrict user privileges to the minimum necessary. * **Purpose:** Ensures that users only access resources necessary for their role. |
| Accounting | * **Definition:** Tracks and logs user activities for auditing and compliance. * **Application:** Log all login attempts, data changes, and resource accesses. * **Purpose:** Enables traceability and accountability for security audits and incident investigations. |

**\***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.

|  |  |  |
| --- | --- | --- |
| **Standard ID** | **Principles** | **Justification** |
| STD-001-CPP | Validate Input Data, Heed Compiler Warnings | Ensures strong typing prevents unintended type conversions and addresses compiler warnings. |
| STD-002-CPP | Validate Input Data, Use Effective Quality Assurance Techniques | Ensures return values are handled properly to prevent undefined behavior and enable robust testing. |
| STD-003-CPP | Validate Input Data, Practice Defense in Depth | Null-terminated strings prevent buffer overflows, reinforcing secure boundaries. |
| STD-004-CPP | Validate Input Data, Default Deny | Prepared statements prevent SQL injection by disallowing dynamic query construction. |
| STD-005-CPP | Practice Defense in Depth, Use Effective Quality Assurance Techniques | Prevents resource exhaustion by releasing memory in all execution paths. |
| STD-006-CPP | Heed Compiler Warnings, Keep It Simple | Avoiding assertions for error handling ensures production stability and simplifies code. |
| STD-007-CPP | Use Effective Quality Assurance Techniques, Keep It Simple | Catching exceptions by reference avoids slicing and improves exception handling clarity. |
| STD-008-CPP | Practice Defense in Depth, Keep It Simple | Ensures consistent resource management using RAII to avoid resource leaks. |
| STD-009-CPP | Use Effective Quality Assurance Techniques, Practice Defense in Depth | Properly handling exceptions prevents undefined behavior and adds resilience against unexpected errors. |
| STD-010-CPP | Practice Defense in Depth, Use Effective Quality Assurance Techniques | Synchronization mechanisms ensure safe multithreaded access, avoiding data races and undefined behavior. |

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 11/12/2024 | Added Milestone Content | Victor Udeh |  |
| 1.2 | 12/06/2024 | Added Project 1 content | Victor Udeh |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

Green Pace Secure Development Policy

**Overview** Software development at Green Pace requires consistent implementation of secure principles for 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 core security principles, C/C++ coding standards, authorization, authentication, auditing standards, and data encryption standards. It explains the differences between policy, standards, principles, and practices.

**Scope** This document applies to all staff creating, deploying, or supporting custom software at Green Pace.

**Ten Core Security Principles**

1. **Validate Input Data**: Always validate data received from outside sources to prevent injection attacks or data corruption. This principle ensures data integrity and prevents vulnerabilities like buffer overflows and SQL injection.
2. **Heed Compiler Warnings**: Treat compiler warnings as errors and address them immediately. Warnings are often indicators of code that could lead to undefined behavior, making the application vulnerable.
3. **Architect and Design for Security Policies**: Incorporate security requirements into the design phase to prevent vulnerabilities at the structural level. Proper architectural design ensures a solid foundation for secure coding.
4. **Keep It Simple**: Complexity is the enemy of security. Simple and clean code is less prone to errors and easier to maintain, review, and secure.
5. **Default Deny**: By default, access should be denied and permissions explicitly granted. This prevents unauthorized access by ensuring that only approved actions are allowed.
6. **Adhere to the Principle of Least Privilege**: Limit permissions to only what is necessary for a specific function. This reduces the risk of damage or misuse if an account or process is compromised.
7. **Sanitize Data Sent to Other Systems**: Always sanitize data that is sent between different components or external systems. This practice prevents injection attacks and data leakage.
8. **Practice Defense in Depth**: Implement multiple layers of defense so that if one layer is compromised, others can still protect the system. This provides redundancy in security controls.
9. **Use Effective Quality Assurance Techniques**: Employ thorough testing, code reviews, and static analysis to identify security vulnerabilities early in the development cycle.
10. **Adopt a Secure Coding Standard**: Use a standard such as SEI CERT C++ Coding Standard to ensure consistent and secure practices throughout the development process.

**C/C++ Ten Coding Standards**

**Coding Standard 1: Data Type**

* **Label**: STD-001-CPP
* **Name of Standard**: Use Strong Typing for Data Types
* **Rationalize the Standard**: Using strong typing reduces unintended data conversion and prevents bugs related to type misuse.
* **Noncompliant Code**:

int value = "123"; // Implicit conversion may cause runtime issues

**Noncompliant Code Description**: The use of implicit conversion here can lead to unexpected behavior if the input type changes.

* **Compliant Code**:

std::string value = "123"; // Explicit type declaration prevents misuse

**Compliant Code Description**: Using an explicit type that matches the intended data eliminates conversion risks.

**Coding Standard 2: Data Value**

* **Label**: STD-002-CPP
* **Name of Standard**: Limit Use of Magic Numbers
* **Rationalize the Standard**: Magic numbers reduce code readability and make it harder to modify or debug values.
* **Noncompliant Code**:

int timeout = 3000; // Magic number without explanation

**Noncompliant Code Description**: The hardcoded value lacks context, making future changes or debugging difficult.

* **Compliant Code**:

const int TIMEOUT\_MS = 3000; // Constant value provides context

**Compliant Code Description**: Defining the value as a constant provides better understanding and easier maintenance.

**Coding Standard 3: String Correctness**

* **Label**: STD-003-CPP
* **Name of Standard**: Avoid Buffer Overflow with Strings
* **Rationalize the Standard**: Ensuring that strings do not exceed buffer limits prevents buffer overflows, which could lead to security vulnerabilities.
* **Noncompliant Code**:
* char buffer[10];

strcpy(buffer, "This string is too long");

**Noncompliant Code Description**: This code may cause a buffer overflow due to the lack of boundary checking.

* **Compliant Code**:
* char buffer[10];
* strncpy(buffer, "This string", sizeof(buffer) - 1);

buffer[sizeof(buffer) - 1] = '\0';

**Compliant Code Description**: Using strncpy with size limits prevents overflowing the buffer.

**Coding Standard 4: SQL Injection**

* **Label**: STD-004-CPP
* **Name of Standard**: Use Parameterized Queries
* **Rationalize the Standard**: Using parameterized queries prevents SQL injection by separating code from data.
* **Noncompliant Code**:

std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'";

**Noncompliant Code Description**: Concatenating strings for SQL queries makes the code vulnerable to SQL injection.

* **Compliant Code**:
* std::string query = "SELECT \* FROM users WHERE name = ?";

preparedStatement.setString(1, userInput);

**Compliant Code Description**: Parameterized queries ensure that user input is safely handled without risk of injection.

**Coding Standard 5: Memory Protection**

* **Label**: STD-005-CPP
* **Name of Standard**: Free Allocated Memory
* **Rationalize the Standard**: Not freeing allocated memory leads to memory leaks and potential Denial of Service.
* **Noncompliant Code**:
* int\* ptr = new int(5);

// No delete called, memory leak occurs

**Noncompliant Code Description**: Forgetting to free the memory can result in memory leaks and resource exhaustion.

* **Compliant Code**:
* int\* ptr = new int(5);

delete ptr;

**Compliant Code Description**: Freeing allocated memory ensures no resource leaks.

**Coding Standard 6: Assertions**

* **Label**: STD-006-CPP
* **Name of Standard**: Use Assertions for Critical Checks
* **Rationalize the Standard**: Assertions allow critical errors to be caught during development, avoiding undefined runtime behavior.
* **Noncompliant Code**:
* if (value < 0) {
* // Assume it's always positive

}

**Noncompliant Code Description**: No mechanism is present to detect unexpected values, which may lead to faulty logic.

* **Compliant Code**:

assert(value >= 0);

**Compliant Code Description**: Assertions enforce critical assumptions during development and debugging.

**Coding Standard 7: Exceptions**

* **Label**: STD-007-CPP
* **Name of Standard**: Properly Handle Exceptions
* **Rationalize the Standard**: Handling exceptions ensures the program can gracefully recover or exit without undefined behavior.
* **Noncompliant Code**:
* try {
* riskyFunction();
* } catch (...) {
* // Swallowing exceptions silently

}

**Noncompliant Code Description**: Swallowing exceptions without handling them leaves the system in an unknown state.

* **Compliant Code**:
* try {
* riskyFunction();
* } catch (const std::exception& e) {
* std::cerr << "Error: " << e.what() << std::endl;

}

**Compliant Code Description**: Logging exceptions allows issues to be properly tracked and diagnosed.

**Coding Standards 8, 9, 10** Provide three additional coding standards relevant to your area, such as additional memory safety practices or data handling techniques, and ensure they are labeled and filled out similarly to the above examples.