**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 | Input validation is a critical security control that ensures only properly formed data is entering the system, preventing malformed data from causing unintended effects. This principal guards against various forms of injection attacks and helps in maintaining the integrity and reliability of the system by rejecting any input that does not meet the predefined criteria for acceptance. |
| 1. Heed Compiler Warnings | Compiler warnings are valuable indicators of potential security flaws or bugs in code. By paying attention to and resolving these warnings, developers can preemptively address issues that may lead to vulnerabilities. This principle emphasizes the importance of not ignoring compiler output, as it can be an early detection tool for preventing security weaknesses. |
| 1. Architect and Design for Security Policies | Security must be integrated into the architecture and design phases of software development. This principle involves designing systems with security as a primary consideration, ensuring that security policies and mechanisms are in place from the outset. It means building systems that are robust against attacks and can enforce security policies effectively. |
| 1. Keep It Simple | Complexity is the enemy of security. The more complex a system, the harder it is to understand, secure, and maintain. This principal advocates for simplicity in design, implementation, and interfaces, reducing the chances of security flaws due to misunderstandings or misconfigurations and making it easier to apply security controls effectively. |
| 1. Default Deny | This principle states that access to resources should be denied by default, and only granted explicitly to authorized entities. It is a foundational concept in access control systems, ensuring that permissions are not given freely but are carefully allocated according to the principle of least privilege. |
| 1. Adhere to the Principle of Least Privilege | Each process, user, or system should have the minimum privileges necessary to perform its tasks, and no more. This principle limits the damage that can be done in the event of an exploit or compromise, as the attacker's capabilities are restricted to only what is necessary for the legitimate purposes of the compromised entity. |
| 1. Sanitize Data Sent to Other Systems | Data exchanged between systems can be an attack vector if not properly sanitized. This principle ensures that data leaving the system is stripped of sensitive information and is formatted correctly to prevent injection attacks on the receiving system. It helps in maintaining data integrity and confidentiality across system boundaries. |
| 1. Practice Defense in Depth | Security should be layered, with multiple defenses covering potential points of attack. Even if one layer of security is breached, additional layers ensure that the system is not compromised. This principle acknowledges that no single security measure is foolproof, and multiple overlapping security controls are necessary for robust protection. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance (QA) is essential in identifying and mitigating security vulnerabilities. This principle emphasizes the use of comprehensive testing, code reviews, static and dynamic analysis tools, and other QA practices to uncover and address security issues before software is deployed. |
| 1. Adopt a Secure Coding Standard | Establishing and following secure coding standards helps prevent common vulnerabilities. This principle involves the use of guidelines that dictate how to write secure code, covering aspects such as input validation, error handling, and cryptography. It encourages a consistent, security-focused approach to coding, reducing the likelihood of introducing vulnerabilities. |

### 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** | **Strong Type Enforcement** |
| --- | --- | --- |
| **Data Type** | [STD-001-DTC] | Enforcing strong type checking helps prevent type conversion errors that can lead to data corruption or undefined behavior. |

| **Noncompliant Code** |
| --- |
| The code unsafely casts an integer pointer to a double pointer, leading to potential misinterpretation of data types. |
| int i;  void\* ptr = &i;  double\* dbl = (double\*) ptr;  \*dbl = 0.0; // Unsafe type casting |

| **Compliant Code** |
| --- |
| The integer is safely converted to a double using static\_cast, ensuring correct type conversion. |
| int i;  double d = static\_cast<double>(i); // Safe casting |

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

| **Principles(s):** Economy of Mechanism  This principle emphasizes simplicity and minimalism. By enforcing strong type checking, the standard reduces complexity and potential errors in the codebase, making the system easier to understand and safer to maintain.  Principle: Fail-Safe Defaults  By default, this standard enforces safer type conversions, reducing the chances of errors. It mandates the use of safe methods like static\_cast over more permissive, error-prone operations like C-style casts, ensuring that operations default to safer, controlled behaviors.  Principle: Defense in Depth  This principle implies having multiple layers of control. Enforcing strong type enforcement adds an additional security layer at the code level, which works in conjunction with other security measures to safeguard the application from data corruption and undefined behaviors that could be exploited. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.0 | C++ Type Safety Rule | This tool checks for unsafe type conversions in C++ code. It flags uses of C-style casts and suggests safer alternatives like static\_cast, dynamic\_cast, etc. |
| Clang Static Analyzer | 12.0.1 | CastChecker | Clang's static analysis tool identifies potentially unsafe casts in C++ applications, providing warnings when implicit conversions might lead to unexpected behaviors. |
| Coverity | 2021 R2 | BAD\_CAST | Coverity's static analysis capabilities include detection of risky type casting practices that could lead to security and stability issues. |
| Cppcheck | 2.8 | typeSafety | Cppcheck is an open-source static code analysis tool for C and C++ languages. It helps in detecting the misuse of C++ types, including dangerous type conversions and casts. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Valid Range Checking** |
| --- | --- | --- |
| **Data Value** | [STD-002-DVC] | Implementing range checking on data values prevents errors and vulnerabilities associated with out-of-range values. |

| **Noncompliant Code** |
| --- |
| The function accesses an array without verifying that the index is within valid bounds. |
| int getIndex(int index, int size) {  return arr[index]; // No bounds checking  } |

| **Compliant Code** |
| --- |
| The function checks if the index is within the bounds of the array size before accessing the array. |
| int getIndex(int index, int size) {  if (index >= 0 && index < size) {  return arr[index];  }  throw std::out\_of\_range("Index out of bounds");  } |

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

| **Principles(s):** Fail-Safe Defaults  This principle emphasizes safe default behaviors to prevent errors. Applying this principle, the standard ensures that any data operation defaults to safety by mandating bounds checking, thereby preventing out-of-bounds access by default.  Principle: Defense in Depth  This principle involves layering security measures. Valid range checking acts as a fundamental layer of security by preventing buffer overflow attacks and ensuring data integrity, which is critical in a layered security strategy.  Principle: Economy of Mechanism  This principle advocates for simplicity and straightforwardness in security mechanisms. Range checking is a simple yet effective control that can prevent a variety of common software vulnerabilities, such as out-of-bounds errors and buffer overflows.  Principle: Complete Mediation  Every access to every resource must be checked for authorization. In the context of data handling, complete mediation means validating every index or pointer access against the allowable bounds, ensuring all data accesses are legitimate and safe. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.0 | ArrayIndexOutOfBounds | This checker scans for instances where array indices might be accessed without prior bounds checking, flagging potential vulnerabilities and suggesting the insertion of range checks. |
| Coverity | 2021 R2 | OUT\_OF\_BOUNDS | Coverity’s analysis includes the detection of out-of-bounds accesses, helping developers identify and fix locations in the code where bounds checking is insufficient or missing. |
| Polyspace Bug Finder | R2021a | Out of Bounds Checker | This tool uses static analysis to identify potential out-of-bounds accesses in C and C++ code before runtime, providing an effective means to catch and correct range-related errors during the development phase. |
| Klocwork | 2021 | ARR\_BOUND | Klocwork provides insights into array boundary violations in code, offering detailed feedback and remediation advice to ensure safe index operations within valid bounds. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Safe String Handling** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | Proper string handling prevents buffer overflow, a common source of security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The code attempts to copy a string that is too long for the buffer, causing an overflow. |
| char buff[10];  strcpy(buff, "This is a long string"); // Buffer overflow |

| **Compliant Code** |
| --- |
| The string is safely copied into the buffer, ensuring that it does not exceed the buffer's limit. |
| char buff[10];  strncpy(buff, "Short", sizeof(buff)); // Safe copying  buff[sizeof(buff) - 1] = '\0'; // Null termination |

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

| **Principles(s):** Fail-Safe Defaults This principle emphasizes ensuring that systems fail in a way that ensures security. For string handling, this involves using safer functions like strncpy instead of strcpy to prevent buffer overflows by default, thus avoiding severe vulnerabilities.  Principle: Defense in Depth  This principle involves using multiple layers of security. Safe string handling provides a fundamental layer by preventing buffer overflow attacks, which are common vectors for more complex exploits.  Principle: Economy of Mechanism  This principle advocates for simple, straightforward, and reliable security measures. By implementing straightforward bounds checking and safe copying routines, this standard simplifies security in string operations, making it less prone to error.  Least Common Mechanism  This principle seeks to minimize the mechanisms shared by different users, providing them with as much isolation as possible. In the context of string handling, it encourages practices that prevent shared buffer overflows affecting multiple system components or users. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 20.2.0 | Buffer Overflow | Fortify analyzes source code to identify potential security vulnerabilities, including unsafe string operations that could lead to buffer overflows, offering recommendations for safer coding practices. |
| Coverity | 2021 R2 | TAINTED\_STRING | Coverity's analysis includes checking for vulnerabilities that arise from improper handling of strings, such as using untrusted input in string operations without adequate checking. |
| Checkmarx | 9.0 | Buffer Overflow | Checkmarx provides comprehensive analysis tools that include checking for buffer overflows caused by unsafe string handling, highlighting lines of code where the vulnerabilities occur and suggesting fixes. |
| Clang Static Analyzer | 12.0.1 | StringSafety | Clang's analyzer can identify potential string-related issues, including buffer overflows and improper null termination, which are critical for maintaining string safety in C/C++ applications. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Preventing SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | Preventing SQL injection is essential for protecting databases from unauthorized access and modification. |

| **Noncompliant Code** |
| --- |
| The query string is constructed by directly including user input, which may contain malicious SQL. |
| std::string query = "SELECT \* FROM users WHERE user\_id = '" + user\_input + "'";  executeQuery(query); // Vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| The query uses prepared statements with parameterized queries, preventing SQL injection. |
| std::string query = "SELECT \* FROM users WHERE user\_id = ?";  PreparedStatement\* stmt = conn.prepareStatement(query);  stmt->setInt(1, user\_input);  stmt->execute(); |

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

| **Principles(s):** Least Privilege  This principle ensures that only necessary permissions are granted. In the context of SQL injection, using parameterized queries or prepared statements minimizes the risk by not allowing direct execution of arbitrary SQL code.  Principle: Defense in Depth  This principle involves multiple layers of security controls. SQL injection prevention is a critical layer that protects against database attacks, supporting other layers such as input validation and regular expressions filtering.  Principle: Complete Mediation  Every access must be checked for authorization. Prepared statements ensure that each query executed against the database is explicitly defined and parameter values are treated only as data, never as part of the SQL command, thus mediating access strictly.  Principle: Fail-Safe Defaults  Systems should default to a secure state. By using prepared statements as the default method for database interaction, applications inherently block SQL injection by design, making security the default state. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP (Zed Attack Proxy) | 2.10.0 | SQL Injection Scanner | OWASP ZAP can automatically scan applications to detect SQL injection vulnerabilities by injecting malicious SQL statements to see if the application is vulnerable. |
| IBM Security AppScan | 10.0.1 | SQL Injection | AppScan provides a dynamic application security testing (DAST) tool that identifies SQL injection vulnerabilities by simulating attacks on applications and analyzing responses. |
| Fortify Static Code Analyzer | 20.2.0 | SQL Injection | Fortify analyzes source code to detect places where SQL injections could occur, suggesting the use of safer coding practices like parameterized queries. |
| SonarQube | 9.0 | SQL Injection Vulnerability | SonarQube performs static analysis to identify unsafe SQL query construction patterns in code, offering remediation advice to switch to safer constructs. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Safe Memory Management** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | Proper memory management prevents leaks and corruption, thereby safeguarding system stability and security. |

| **Noncompliant Code** |
| --- |
| Memory allocated on the heap is not freed, leading to a memory leak. |
| int\* ptr = new int(10);  // Forgot to delete ptr |

| **Compliant Code** |
| --- |
| The use of smart pointers automatically manages the memory, preventing leaks. |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(10); |

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

| **Principles(s):** Economy of Mechanism  This principle emphasizes simplicity and minimalism in design. Using smart pointers simplifies memory management by automating resource deallocation, thereby reducing complexity and the chance of human error.  Principle: Fail-Safe Defaults  Systems should default to secure states. Smart pointers provide a fail-safe mechanism by default, automatically managing memory lifecycle and ensuring that memory is freed when no longer needed.  Principle: Least Privilege  Each module (or user, process, program, etc.) should operate using the least amount of privilege necessary. By managing memory access through smart pointers, applications limit direct memory manipulation, minimizing the potential for harmful actions.  Principle: Resource Encapsulation  Encapsulation limits the exposure of functionality and data, which aligns closely with using smart pointers that encapsulate memory management tasks and reduce direct memory access by developers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.15.0 | Memcheck | Valgrind’s Memcheck tool detects memory management errors such as leaks and incorrect deallocations, essential for ensuring safe memory practices in development. |
| AddressSanitizer (part of LLVM/Clang) | LLVM 10.0 | AddressSanitizer | This tool is integrated into compilers and provides runtime memory corruption detection, effective in catching use-after-free, buffer overflows, and other memory issues. |
| Dr. Memory | 2.3.0 | Memory Leak Detector | Dr. Memory is a memory monitoring tool capable of identifying and reporting memory leaks and allocation errors in applications, which is pivotal for maintaining robust memory management practices. |
| Microsoft Visual Studio | 2022 | C++ Core Guidelines Checker | This toolset within Visual Studio provides guidelines enforcement for modern C++, including rules aimed at preventing common memory management errors by advocating the use of modern language features like smart pointers. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use of Assertions for Internal State Checking** |
| --- | --- | --- |
| **Assertions** | [STD-006-AST] | Assertions help catch internal errors and inconsistencies during development, preventing them from causing harm in production. |

| **Noncompliant Code** |
| --- |
| The code does not assert the divisor is non-zero, potentially leading to division by zero. |
| int calculate(int divisor) {  return 100 / divisor; // No check for divisor being zero  } |

| **Compliant Code** |
| --- |
| The assertion ensures the divisor is not zero, preventing division by zero errors. |
| int calculate(int divisor) {  assert(divisor != 0);  return 100 / divisor;  } |

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

| **Principles(s):** Fail-Safe Defaults  Assertions are a developmental tool designed to fail fast during the development cycle if an assumption is violated, thereby ensuring that unsafe code does not proceed into production undetected.  Principle: Defense in Depth  Assertions provide an additional layer of verification during development, ensuring that internal states are consistent and expected behaviors are met before the software is deployed.  Principle: Complete Mediation  By verifying every critical operation's preconditions through assertions, it ensures that all operations are executed only with correct and expected inputs, enhancing overall system integrity.  Principle: Economy of Mechanism  Assertions provide a simple and straightforward mechanism to check and validate assumptions continuously and consistently across the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analysis Tools (General Category) | N/A | Assertion Checker | Most static code analysis tools will check for the presence of assertions in the code where critical variables and state conditions are manipulated, helping ensure that these assumptions are explicitly verified. |
| Clang Static Analyzer | 12.0.1 | AssertUse | This tool checks for proper use of assertions in C/C++ programs to ensure that potential run-time errors can be caught during development. |
| Visual Studio Code Analysis | 2022 | C++ Best Practice Checks | Visual Studio's built-in code analysis tools include checks for adequate assertion use, verifying that developers are asserting conditions that could lead to high-risk vulnerabilities. |
| SonarQube | 9.0 | S2583 - Assertions should not be used on boolean constants | SonarQube not only ensures assertions are used correctly but also that they are used effectively, preventing the misuse of assertions that could lead to ignored code paths in testing. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Robust Exception Handling** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXP] | Proper exception handling ensures that errors are caught and handled gracefully, maintaining application stability and security. |

| **Noncompliant Code** |
| --- |
| The exception is caught but the resource is not properly managed in the case of an error. |
| try {  int\* ptr = new int[10];  // Code that may throw  delete[] ptr;  } catch (...) {  // Exception caught but not handled correctly  } |

| **Compliant Code** |
| --- |
| The STL container manages memory automatically, and exceptions are handled explicitly. |
| try {  std::vector<int> vec(10);  // Code that may throw  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  // Properly handle the exception  } |

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

| **Principles(s):** Fail-Safe Defaults  This principle ensures that if an error occurs, the system defaults to a secure state. Robust exception handling encapsulates error handling within the program flow, ensuring that exceptions do not lead to unstable states or security vulnerabilities.  Principle: Defense in Depth  Exception handling adds a layer of protection by managing unexpected conditions without crashing the system, thereby maintaining the integrity and availability of the application even under erroneous conditions.  Principle: Least Privilege  By handling exceptions properly, the system ensures that processes operate with only the necessary access needed at that time, especially during error states, which might otherwise expose sensitive information or system capabilities.  Principle: Economy of Mechanism  Exception handling should be simple yet comprehensive. By managing exceptions through centralized handling mechanisms and using standard libraries like STL, the system minimizes complexity and reduces the chance of error in the handling logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.0 | Exception Handling Rules | SonarQube checks for common pitfalls in exception handling in various programming languages, ensuring that exceptions are caught and handled properly, and resources are safely managed. |
| Coverity | 2021 R2 | BAD\_EXCEPT\_HANDLING | Coverity detects issues in exception handling paths that may lead to resource leaks, crashes, or inconsistent states, offering guidance for more robust practices. |
| ReSharper C++ | 2021.2 | C++ Exception Handling | ReSharper provides insights and automatic corrections for exception handling in C++, ensuring that all potential code paths are covered, and exceptions are managed correctly. |
| Visual Studio | 2022 | Exception Assistant | Visual Studio's Exception Assistant helps debug exceptions when they occur, providing real-time insights into the context of the exception and suggestions for improvement in handling strategies. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Proper Use of Synchronization Mechanisms** |
| --- | --- | --- |
| Synchronization | [STD-008-SYN] | Proper synchronization is crucial in multi-threaded applications to prevent race conditions, deadlocks, and other concurrency issues. |

| **Noncompliant Code** |
| --- |
| Incrementing a shared resource without synchronization can lead to race conditions. |
| int shared\_resource;  void increment() {  shared\_resource++; // Noncompliant: No synchronization  } |

| **Compliant Code** |
| --- |
| The use of std::lock\_guard ensures that the shared resource is accessed by only one thread at a time, preventing race conditions. |
| #include <mutex>  std::mutex mtx;  int shared\_resource;  void increment() {  std::lock\_guard<std::mutex> lock(mtx);  shared\_resource++; // Compliant: Resource is protected  } |

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

| **Principles(s):** Defense in Depth  This principle involves implementing multiple layers of security to safeguard the system. Synchronization mechanisms act as a fundamental security layer in multi-threaded environments, preventing race conditions and ensuring that operations on shared resources are handled safely.  Principle: Least Privilege  This principle ensures that processes have only the privileges they need to perform their tasks, and no more. In the context of synchronization, it means that access to shared resources is granted to only one thread at a time, minimizing the risk of unintended interactions.  Principle: Fail-Safe Defaults  By default, operations involving shared resources should be safe and secure. Using std::lock\_guard ensures that every access is automatically locked and released, providing a fail-safe approach to resource management. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Meidum | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helgrind (part of Valgrind) | 3.15.0 | Race Condition Detector | Helgrind detects race conditions in C/C++ applications using POSIX threads, helping identify where synchronization may be missing or incorrectly implemented. |
| Intel Inspector | 2020 | Threading Error Analysis | This tool identifies synchronization issues such as deadlocks and race conditions, providing detailed diagnostics to help developers correct threading errors. |
| Coverity | 2021 R2 | CONCURRENCY | Coverity’s static analysis detects potential threading issues and synchronization errors, suggesting improvements to ensure thread safety. |
| SonarQube | 9.0 | Synchronization on Static Fields | SonarQube checks for common pitfalls in synchronization, such as synchronizing on static fields which can lead to deadlock scenarios or ineffective locking practices. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Secure Network Data Handling** |
| --- | --- | --- |
| Network Programming | [STD-009-SNP] | Ensuring data received over the network is validated and sanitized prevents a wide range of vulnerabilities, such as injection attacks and data corruption. |

| **Noncompliant Code** |
| --- |
| Data received from a network is processed without validation, potentially leading to security issues. |
| char buffer[1024];  recv(socket, buffer, sizeof(buffer), 0); // Noncompliant: No validation  processData(buffer); |

| **Compliant Code** |
| --- |
| Data is validated and properly null-terminated before processing, enhancing security. |
| char buffer[1024];  int length = recv(socket, buffer, sizeof(buffer) - 1, 0);  if (length > 0) {  buffer[length] = '\0'; // Ensure null termination  if (validateData(buffer)) { // Validate before processing  processData(buffer);  }  } |

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

| **Principles(s):** Complete Mediation  This principle ensures that all access to a resource is checked for authorization and appropriateness. Applying it to network data handling means every piece of data received over the network is verified and sanitized before processing, preventing unauthorized or malicious data from causing harm.  Principle: Fail-Safe Defaults  Systems should default to secure states. For network data handling, this involves validating all incoming data as unsafe until proven otherwise, ensuring that only properly validated and sanitized data is processed.  Principle: Least Privilege  This principle dictates that operations should have only the minimum privileges necessary to perform their function. In the context of network data, this means processing the smallest set of data necessary and limiting how much data each part of the program can access and manipulate. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Wireshark | 3.4.5 | Protocol Analyzer | Wireshark can monitor network traffic and highlight potential issues in data packets, including malformed data or suspicious patterns that might indicate an attack. |
| Snort | 3.0 | Network Intrusion Detection | Snort is an open-source network intrusion detection system that can detect various types of attacks and vulnerabilities by analyzing network traffic patterns. |
| IBM Security AppScan | 10.0.1 | Network Data Validation | AppScan provides dynamic application security testing (DAST) that includes checking for vulnerabilities in how applications handle network data, ensuring that data is correctly validated and sanitized. |
| OWASP ZAP | 2.10.0 | Network Data Analyzer | ZAP can act as a proxy to examine the data being sent and received by applications, checking for data validation issues and offering suggestions for secure handling practices. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Thread-Safe Practices** |
| --- | --- | --- |
| Thread Safety | [STD-010-TFS] | Writing thread-safe code is essential for ensuring correct program execution in a multi-threaded environment. |

| **Noncompliant Code** |
| --- |
| Modifying a shared std::vector without synchronization can lead to data corruption. |
| std::vector<int> shared\_vector;  void unsafe\_add(int value) {  shared\_vector.push\_back(value); // Noncompliant: Vector is not thread-safe  } |

| **Compliant Code** |
| --- |
| The use of a mutex ensures that the vector is modified by only one thread at a time, making the operation thread-safe. |
| #include <mutex>  std::mutex vector\_mutex;  std::vector<int> shared\_vector;  void safe\_add(int value) {  std::lock\_guard<std::mutex> lock(vector\_mutex);  shared\_vector.push\_back(value); // Compliant: Synchronized access  } |

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

| **Principles(s):** Defense in Depth  This principle involves implementing multiple layers of security to safeguard the system. In multi-threaded programming, using mutexes and other synchronization tools provides a fundamental layer of safety, ensuring that data is accessed in a controlled and safe manner, preventing race conditions and deadlocks.  Principle: Least Privilege  Each process or thread should have only the minimal amount of access necessary. Applying mutex locks to shared resources ensures that only one thread can access a resource at a time, minimizing the risk of concurrent access and related issues.  Principle: Economy of Mechanism  This principle advocates for simple, straightforward, and reliable security mechanisms. Using well-understood and widely adopted synchronization primitives like mutexes simplifies the development of multi-threaded applications and reduces the likelihood of errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helgrind (part of Valgrind) | 3.15.0 | Race Condition Detector | Helgrind detects race conditions in C/C++ applications using POSIX threads, helping identify where synchronization may be missing or incorrectly implemented. |
| Intel Inspector | 2020 | Threading Error Analysis | This tool identifies synchronization issues such as deadlocks and race conditions, providing detailed diagnostics to help developers correct threading errors. |
| Coverity | 2021 R2 | THREAD\_SAFETY\_VIOLATIONS | Coverity’s static analysis detects potential threading issues and synchronization errors, suggesting improvements to ensure thread safety. |
| ThreadSanitizer | Included with GCC and Clang | Thread Safety Analysis | ThreadSanitizer is a runtime tool that detects data races and other threading-related issues, providing reports that help identify the specific locations and conditions causing these problems. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Assess and Plan

Automation starts with integrating threat intelligence and risk assessment tools that automatically update threat models and backlog priorities. Tools can scan code repositories and match them against known vulnerabilities to inform planning.

Design

Security-as-code methodologies can be implemented, with automated tools checking design patterns against security best practices. Automated compliance checks against standards like OWASP can ensure designs are secure by default.

Build

Incorporate automated secure code analysis tools in the CI pipeline to scan and identify security issues upon each commit. Ensure that only code passing these checks is merged into the build. Automation can also enforce the use of trusted repositories and validate open-source dependencies for known vulnerabilities.

Verify and Test

Utilize automated vulnerability scanning and testing tools to ensure the code behaves securely under various conditions. Automated compliance scanning can ensure that the software adheres to legal and regulatory standards.

Pre-Production

Before deployment, automate the process of configuration and health checks, including penetration testing. Configuration management tools can ensure the deployment environment adheres to the defined security posture.

Transition and Health Check

When transitioning to production, automate the deployment process with pre-defined security settings. This can include automated scripts to apply necessary security configurations and conduct health checks.

Production

Once in production, automate real-time security monitoring with SIEM tools to log and alert on potential security events. Automation can facilitate continuous compliance and intrusion detection to identify and alert on active security threats.

Monitor and Detect

In this phase, automation is key for monitoring system health and detecting anomalies. Anomaly detection systems can learn normal behavior and trigger alerts when deviations occur, enabling quicker response times.

Maintain and Stabilize

Post-incident, automated tools should assess the impact against a security baseline and assist in returning to a stable state. This might involve automated patching tools and configuration scripts that can restore systems to their pre-attack configuration.

Respond

Automate response actions to contain breaches. This can include isolating affected systems, blocking malicious traffic, and potentially rolling back systems to a secure state.

Learn and Adapt

Finally, use automation for continuous learning by analyzing incident reports and audit logs to improve the security posture. Machine learning can identify patterns and suggest improvements to the security policy and controls.

In every phase of this cycle, automation not only enforces compliance and security measures but also significantly reduces the manual overhead, speeding up the entire process and enabling the security team to focus on more strategic tasks. Implementing such comprehensive automation ensures that security is not a one-time checkpoint but a continuous and integral part of the entire pipeline.

### 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-DVC | High | High | Medium | High | 2 |
| STD-003-STR | High | High | Medium | High | 2 |
| STD-004-SQL | High | High | Medium | High | 2 |
| STD-005-MEM | High | High | Medium | High | 2 |
| STD-006-AST | High | Medium | Low to Medium | High | 2 |
| STD-007-EXP | High | Medium | Medium | High | 2 |
| STD-008-SYN | High | High | Medium | High | 2 |
| STD-009-SNP | High | High | Medium | High | 2 |
| STD-010-TFS | High | High | Medium | High | 2 |
| [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 | Encryption at rest involves securing data that is stored on physical media, preventing unauthorized access when the data is not actively being used.  This policy is crucial for protecting sensitive data from unauthorized access, particularly in the event of physical security breaches, theft, or unauthorized data access during maintenance and support. It should be enforced whenever data is written to storage media. |
| Encryption in flight | Encryption in flight is concerned with protecting data as it is transmitted across networks.  It is essential to safeguard data from interception, eavesdropping, and man-in-the-middle attacks during transmission. This policy applies whenever data is sent over any network, particularly the internet. |
| Encryption in use | Encryption in use protects data that is being processed by ensuring it remains encrypted during its lifecycle.  Essential for protecting sensitive information from unauthorized access during processing phases, especially in cloud computing environments and third-party processing scenarios. This should be applied whenever data is actively being used in computations or manipulations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of a user, device, or entity in the system, typically using credentials like passwords, tokens, or biometric data.  Authentication is critical for ensuring that system access is granted only to verified users. It applies whenever a user attempts to access the system, ensuring that all user identities are verified before access is allowed. |
| Authorization | Authorization involves granting or denying rights and permissions to resources based on authenticated entities.  Authorization ensures that authenticated users have appropriate permissions. This is crucial for preventing unauthorized access to sensitive data and functions within the system. It applies any time a user interacts with the system’s resources. |
| Accounting | Accounting tracks and logs user activities, providing an audit trail that can be used to reconstruct events, detect breaches, and monitor user actions.  Accounting is vital for non-repudiation, helping trace actions to individual users and identifying the source of issues or breaches. It applies continuously as it must capture all significant actions and changes throughout the system to provide a clear trail for audits and 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.

**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 | 04/15/2024 | Updated | Nathan Wilson | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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