**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

Mary Vuong

## 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 helps to ensure the safety and correctness of any application. This principle involves checking every piece of data provided by users to make sure it meets the expected format, type, and length, and is free from potentially harmful content. Effective input validation can prevent many types of security issues, such as SQL injections and cross-site scripting, which occur when unexpected data manipulates the system. |
| 1. Heed Compiler Warnings | Compilers often provide warnings that can flag potential security or functional issues in code. By paying attention to these warnings and addressing them proactively, developers can prevent many bugs and vulnerabilities from making it into the final software product. This practice supports maintaining high code quality. |
| 1. Architect and Design for Security Policies | This principle emphasizes the importance of incorporating security considerations into the architecture and design phases of software development. By planning for security from the beginning, developers can make sure that the software complies with necessary security policies and standards, and that protective measures are built into the system, rather than added as an afterthought. |
| 1. Keep It Simple | Over-complexity can cause many issues with security. Simpler systems are easier to understand, manage, and secure. By reducing complexity, the chances of introducing security flaws are minimized. This principle advocates for straightforward designs and solutions that accomplish goals with the least amount of complexity necessary. |
| 1. Default Deny | In security management, it’s safer to deny all access by default and only allow it through explicit permissions. This principle reduces the risk of unintended access or actions within the system. By only enabling specific functionalities and access rights that are required for legitimate purposes, the system remains secure against many types of threats. |
| 1. Adhere to the Principle of Least Privilege | This security principle involves giving users or systems the minimum levels of access, or permissions, necessary to perform their tasks. This limits the potential damage that could occur from a security breach or from misuse of access privileges. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to different systems, it’s important to cleanse the data of any elements that could be harmful or that are not strictly needed. This includes stripping out potentially executable code, removing unnecessary data that might expose sensitive information, and checking that the data conforms to the expectations of the receiving system. |
| 1. Practice Defense in Depth | This principle conveys to not rely on a single security measure. Instead, layer multiple, overlapping protections throughout the system. If one layer fails, others still stand. This approach includes using firewalls, antivirus software, secure coding practices, and more, to provide a comprehensive shield against various types of cyber threats. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance (QA) is essential in the software development lifecycle to ensure that the application meets the desired standards of quality, including security. Effective QA techniques involve thorough testing, code reviews, and other methodologies that aim to identify and fix defects and security vulnerabilities before the software is deployed. |
| 1. Adopt a Secure Coding Standard | To maintain consistency and improve security across all software projects, it’s preferred for developers to adopt and adhere to a recognized secure coding standard. These standards provide guidelines on best practices and methods to avoid common security pitfalls. |

### 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 | Never qualify a reference type with const or volatile  In C++, const qualifications should be applied correctly to maintain type safety and predictability. The standard specifies that references themselves cannot be const qualified; only the objects they refer to can be. Adhering to this guideline helps prevent undefined behavior and compiler errors. |

| **Noncompliant Code** |
| --- |
| This example shows incorrect application of const to a reference type, which C++ does not allow and results in undefined behavior. |
| #include <iostream>  void f(char c) {  char &const p = c; // Incorrect: const qualifier on reference type  p = 'p'; // Attempt to modify, leading to undefined behavior  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This example adheres to the standard by correctly declaring a reference to a const-qualified char, ensuring no unintended modifications are made. |
| #include <iostream>  void f(char c) {  const char &p = c; // Correct: reference to a const-qualified char  // p = 'p'; // Error if uncommented: read-only variable is not assignable  std::cout << c << std::endl;  } |

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

| **Principles(s):** Safe Type Handling  This principle ensures that data types are managed correctly to avoid type-related errors, preserving software integrity and reliability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low - Resolution involves minor code changes. | P3 - Important but not critical for immediate correction. | L3 - Lower risk; basic checks required. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 3.9 | Checks for const-qualified reference types and produces errors. | Clang includes static analysis to prevent incorrect const-qualification of reference types during development. |
| SonarQube C/C++ Plugin | 4.10 | S3708 | SonarQube performs code quality checks to detect unsafe type usage in C++, such as incorrect const-qualification. |
| Polyspace Bug Finder | R2024a | CERT C++: DCL52-CPP | Polyspace conducts thorough checks against CERT C++ rules, focusing on reference and type safety diagnostics. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not cast to an out-of-range enumeration value  In C++, casting an integer to an enumeration type without making sure that the value is within the range of the enumeration can lead to undefined behavior. This standard emphasizes the importance of validating that a value is within the allowable range before performing the cast. This practice helps prevent errors and ensures that the program behaves predictably under all conditions. |

| **Noncompliant Code** |
| --- |
| This example shows a noncompliant approach where the check for the enumeration range is incorrectly done after the cast, potentially leading to undefined behavior if the integer value is outside the defined range of the enumeration. |
| #include <iostream>  enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar); // Incorrect: Casting before checking the range  if (enumVar < First || enumVar > Third) { // Error: The range check is ineffective after the cast  std::cout << "Error: Out of range" << std::endl;  }  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures the integer value is within the enumeration range before casting, thereby guaranteeing that the cast does not produce undefined behavior. |
| #include <iostream>  enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) { // Correct: Checking range before casting  std::cout << "Error: Out of range" << std::endl;  } else {  EnumType enumVar = static\_cast<EnumType>(intVar); // Safe casting  std::cout << "Value is within range" << std::endl;  }  } |

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

| **Principles(s):** Range Validation  This principle asserts that values, especially those used in type conversions and enumerations, must be explicitly checked to be within the defined valid range to prevent undefined behaviors and ensure data integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 - Necessary but less critical compared to direct security vulnerabilities. | L3 - Moderate risk; requires standard precautions. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | cast-integer-to-enum | Partly checks for proper integer to enumeration casts, ensuring they are within range. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | Analyzes code to ensure that integer to enumeration casts do not exceed defined limits. |
| Helix QAC | 2024.1 | C++3013 | Detects cases where integers are cast to enumeration types outside of their valid range. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | Ensures that expressions with enum underlying type correspond only to valid enumerators. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator  This coding standard ensures that buffers allocated for string data are sufficiently sized to prevent buffer overflows, which are common vulnerabilities in C++ programs. Buffer overflow vulnerabilities can lead to arbitrary code execution, system crashes, and other security issues. Proper handling includes using mechanisms that ensure the storage automatically adjusts to the data size or carefully managing buffer sizes manually. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates potential buffer overflow due to unbounded input, which could exceed the allocated buffer size. |
| #include <iostream>  void f() {  char buf[12]; // Fixed size buffer  std::cin >> buf; // No check to ensure input fits within the buffer  } |

| **Compliant Code** |
| --- |
| This compliant solution uses std::string, which manages memory automatically and ensures that data does not exceed the allocated buffer size, thereby preventing buffer overflow. |
| #include <iostream>  #include <string>  void f() {  std::string input; // std::string manages its memory  std::cin >> input; // Safe input: std::string adjusts size as needed  } |

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

| **Principles(s):** Memory Safety  This principle emphasizes the importance of managing memory resources properly, especially in managing string data to prevent buffer overflows, which are critical for ensuring program stability and security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium - Fixing buffer overflows may require significant refactoring to implement safer string handling practices. | P1 - Immediate attention required due to high impact on application security. | L1 - High risk; prevention is critical for secure application operation |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stream-input-char-array | Checks and warns about potential buffer overflows when using character arrays for input. |
| CodeSonar | 8.1p0 | MISC.MEM.NTERM, LANG.MEM.BO, LANG.MEM.TO | Detects lack of null terminator and potential buffer and type overruns. |
| Helix QAC | 2024.1 | C++5216, DF2835, DF2836, DF2839 | Identifies risks of buffer overflows, particularly with improper string handling. |
| Klocwork | 2024.1 | NNTS.MIGHT, NNTS.TAINTED, NNTS.MUST, SV.UNBOUND\_STRING\_INPUT.CIN | Monitors for unbounded string inputs and ensures string operations are bound within safe limits. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Do not alternately input and output from a file stream without an intervening positioning call  Managing file streams securely requires careful handling of input and output operations to avoid undefined behavior. Similar principles apply to handling SQL queries, where improperly managed input sequences can lead to SQL injection vulnerabilities. Ensuring a proper sequence in handling data, such as using parameterized queries in SQL or positioning calls in file streams, prevents potential security risks. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates improper management of file stream operations, where output is followed directly by input without an intervening positioning call, leading to undefined behavior. |
| #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Output some data"; // Output to file  std::string str;  file >> str; // Direct input without positioning, leading to undefined behavior  } |

| **Compliant Code** |
| --- |
| This compliant solution introduces a file positioning call between output and input operations, ensuring the file stream is in a valid state before subsequent I/O operations, which eliminates the undefined behavior. |
| #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Output some data"; // Output to file  file.seekg(0, std::ios::beg); // Properly repositioning the file stream  std::string str;  file >> str; // Safe input after positioning  } |

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

| **Principles(s):** Proper Sequence Handling  This principle stresses the importance of maintaining a correct sequence in data handling to avoid state inconsistencies and security vulnerabilities, akin to parameterizing SQL queries to prevent SQL injection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium - Correcting these issues may require significant changes in how file operations are handled. | P6 - Important to address, though it may not always result in critical vulnerabilities. | L2 - Moderate risk; requires attention to maintain proper application function. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-FIO50 | Ensures file stream operations are separated by appropriate positioning calls to prevent undefined behavior. |
| CodeSonar | 8.1p0 | IO.IOWOP, IO.OIWOP | Detects potential issues where input and output operations on streams are performed without intervening positioning calls. |
| Helix QAC | 2024.1 | DF4711, DF4712, DF4713 | Monitors file stream usage for correct sequencing of input and output operations. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-FIO50-a | Checks for alternating input and output operations in file streams without the necessary intervening flush or positioning calls. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access freed memory  Accessing memory after it has been freed can lead to undefined behavior, including security vulnerabilities such as use-after-free attacks. These vulnerabilities can potentially be exploited to execute arbitrary code. Proper management of memory lifetimes and ensuring that pointers to freed memory are not dereferenced are important for maintaining software security and stability. |

| **Noncompliant Code** |
| --- |
| This example shows accessing memory after it has been freed, which can lead to undefined behavior and potential security vulnerabilities. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s; // s is deallocated  // ...  s->f(); // Undefined behavior: s is used after it has been freed  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the dynamically allocated memory is not accessed after it has been deallocated. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f(); // Perform operations before deallocating memory  delete s; // Safely deallocate s after use  } |

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

| **Principles(s):** Resource Management  This principle focuses on the proper management of system resources, particularly memory, to prevent errors such as use-after-free, which can lead to serious security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium - Involves thorough checks and potential refactoring of legacy code to introduce safer memory management practices. | P2 - High priority due to the severe implications of memory corruption. | L1 - High risk requiring immediate mitigation strategies. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use | Detects uses of pointers referencing memory that has been freed. |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete, clang-analyzer-alpha.security.ArrayBoundV2 | Analyzes code for improper use of dynamic memory, including use after free scenarios. |
| CodeSonar | 8.1p0 | ALLOC.UAF | Identifies use after free vulnerabilities, helping to prevent security breaches that exploit freed memory. |
| Helix QAC | 2024.1 | C++4303, C++4304 | Provides checks for proper memory usage and flags use after free errors. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Value-returning functions must return a value from all exit paths  All paths in a value-returning function must explicitly return a value to prevent undefined behavior. This approach helps avoid runtime errors and improves both the reliability and maintainability of the code. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example fails to return a value on all paths, leading to undefined behavior if the input is non-negative. |
| int absolute\_value(int a) {  if (a < 0) {  return -a; // Returns only for negative a  }  // Missing return statement for non-negative a  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that a value is returned regardless of the input's sign, thus avoiding undefined behavior. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a; // Ensures a value is returned for non-negative a  } |

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

| **Principles(s):** Code Completeness  Ensures every function path completes as expected by returning a value where required, thus preventing undefined behavior and enhancing the reliability and maintainability of the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium - Fixing these issues may require reviewing and updating multiple functions to ensure compliance. | P8 - Important to address, as it impacts program stability and security. | L2 - Requires moderate attention to mitigate risks effectively. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | return-implicit | Fully checks and flags functions where implicit returning may occur without explicit return statements. |
| Clang | 3.9 | -Wreturn-type | Warns about missing return statements; however, it may not catch all instances, such as those in function-try-blocks. |
| CodeSonar | 8.1p0 | LANG.STRUCT.MRS, LANG.STRUCT.NVNR | Identifies functions where the return statement is missing and functions marked as noreturn that incorrectly return a value. |
| Helix QAC | 2024.1 | DF2888 | Detects missing return statements in functions, ensuring compliance with best practices for safe and predictable code execution. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions  Proper exception handling ensures that no part of the program fails silently or crashes unexpectedly. Handling all exceptions allows for controlled unwinding of the stack, ensuring that destructors are called and resources are released properly. This practice not only prevents potential resource leaks and system instability but also guards against common security vulnerabilities like denial-of-service attacks. |

| **Noncompliant Code** |
| --- |
| This noncompliant example demonstrates a failure to catch exceptions, which results in calling std::terminate() if an exception is thrown, potentially leading to abrupt termination without cleaning up resources. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func(); // Exception thrown here is not caught within f()  }  int main() {  f(); // Also not caught in main(), leading to std::terminate()  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that all exceptions thrown are caught, which allows for proper cleanup and controlled application shutdown, preventing abrupt termination. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func(); // Potential exception thrown  }  int main() {  try {  f();  } catch (...) { // Catch all exceptions  // Handle error, cleanup resources  }  } |

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

| **Principles(s):** Error Handling  This principle emphasizes the importance of managing all exceptions to maintain application stability and prevent abrupt terminations, which aligns with ensuring comprehensive exception handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium - Identifying and properly handling all possible exceptions can require significant refactoring. | P4 - Needs timely resolution, given its potential impact on application stability and user experience. | L3 - Moderate risk that requires adequate checks and balances to mitigate effectively. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all, early-catch-all | Partly checks for exceptions that may escape from main functions or other critical entry points, ensuring all are caught early. |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Identifies potentially unreachable catch blocks, which could indicate areas where exceptions are not handled. |
| Helix QAC | 2024.1 | C++4035, C++4036, C++4037 | Analyzes exception handling mechanisms to ensure no thrown exception is left uncaught within the application's scope. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR51-a, CERT\_CPP-ERR51-b | Verifies that all exceptions thrown are appropriately caught and handled, adhering to best practices in exception management. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Virtual Function Safety | STD-008-CPP | Do not invoke virtual functions from constructors or destructors  Invoking virtual functions in constructors and destructors can lead to unintended behavior because the object being constructed or destroyed does not yet fully exist or has partially ceased to exist. This practice can lead to bugs and vulnerabilities because the virtual functions invoked may not behave as expected if the object's complete state is not established. Proper management of function calls during construction and destruction is essential for maintaining the integrity and safety of class hierarchies. |

| **Noncompliant Code** |
| --- |
| This example shows a noncompliant approach where a base class constructor and destructor call virtual functions, leading to unintended behaviors as the derived class's methods are not yet available or have been destroyed. |
| struct B {  B() { seize(); }  virtual ~B() { release(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  B::seize(); // Intended to do more but B's constructor is not complete  }  void release() override {  // Release resources then call base, but base's destructor has already run  B::release();  }  }; |

| **Compliant Code** |
| --- |
| This compliant solution avoids calling virtual functions directly from constructors and destructors. Instead, it utilizes private, non-virtual member functions to handle resource management, ensuring that the correct functions are called at the correct times without relying on the virtual function mechanism. |
| class B {  void seize\_mine();  void release\_mine();  public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }  protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };  class D : public B {  void seize\_mine();  void release\_mine();  public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }  protected:  void seize() override {  B::seize();  seize\_mine(); // Proper resource management  }  void release() override {  release\_mine();  B::release(); // Proper resource release  }  }; |

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

| **Principles(s):** Secure Object Lifecycle  This principle stresses the importance of maintaining consistent behavior and state integrity throughout an object's lifecycle, particularly during construction and destruction, to avoid invoking behavior on an incompletely constructed or partially destroyed object. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium - Fixing these issues might require significant refactoring. | P2 - Lower risk to direct security impact. | L3 - Moderate risk; requires code reviews to enforce. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | virtual-call-in-constructor, invalid\_function\_pointer | Fully checks for virtual function calls within constructors and destructors, ensuring that objects are safely and constructed and destroyed. |
| CodeSonar | 8.1p0 | LANG.STRUCT.VCALL\_IN\_CTOR, LANG.STRUCT.VCALL\_IN\_DTOR | Detects virtual function calls from constructors and destructors which can lead to undefined behavior. |
| Helix QAC | 2024.1 | C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282 | Provides comprehensive checks for inappropriate virtual function usage during object construction and destruction phases. |
| C/C++test | 2023.1 | CERT\_CPP-OOP50-a, CERT\_CPP-OOP50-b, CERT\_CPP-OOP50-c, CERT\_CPP-OOP50-d | Helps enforce best practices by checking against virtual function calls in constructors and destructors and managing object type safety. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency Deadlock | STD-009-CPP | Avoid deadlock by locking in a predefined order  Handling concurrency correctly in multi-threaded environments helps prevent deadlocks, where multiple threads prevent each other from progressing by holding onto resources needed by others. Ensuring mutexes are locked in a consistent order across threads eliminates the circular wait condition, one of the four necessary conditions for a deadlock. This practice helps maintain smooth operation and resource management in concurrent applications. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is susceptible to deadlock due to locking mutexes without a predefined order, risking that each thread may hold a lock needed by the other. |
| #include <mutex>  #include <thread>  class BankAccount {  int balance;  public:  std::mutex balanceMutex;  BankAccount() = delete;  explicit BankAccount(int initialAmount) : balance(initialAmount) {}  int get\_balance() const { return balance; }  void set\_balance(int amount) { balance = amount; }  };  int deposit(BankAccount \*from, BankAccount \*to, int amount) {  std::lock\_guard<std::mutex> from\_lock(from->balanceMutex);    if (from->get\_balance() < amount) {  return -1; // Indicate error  }  std::lock\_guard<std::mutex> to\_lock(to->balanceMutex);    from->set\_balance(from->get\_balance() - amount);  to->set\_balance(to->get\_balance() + amount);    return 0;  }  void f(BankAccount \*ba1, BankAccount \*ba2) {  std::thread thr1(deposit, ba1, ba2, 100);  std::thread thr2(deposit, ba2, ba1, 100);  thr1.join();  thr2.join();  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a manual ordering based on the bank account IDs to avoid deadlock. By locking the mutexes in a consistent order, circular wait is prevented. |
| #include <atomic>  #include <mutex>  #include <thread>  class BankAccount {  static std::atomic<unsigned int> globalId;  const unsigned int id;  int balance;  public:  std::mutex balanceMutex;  BankAccount() = delete;  explicit BankAccount(int initialAmount) : id(globalId++), balance(initialAmount) {}  unsigned int get\_id() const { return id; }  int get\_balance() const { return balance; }  void set\_balance(int amount) { balance = amount; }  };  std::atomic<unsigned int> BankAccount::globalId(1);  int deposit(BankAccount \*from, BankAccount \*to, int amount) {  std::mutex \*first;  std::mutex \*second;  if (from->get\_id() < to->get\_id()) {  first = &from->balanceMutex;  second = &to->balanceMutex;  } else {  first = &to->balanceMutex;  second = &from->balanceMutex;  }  std::lock\_guard<std::mutex> firstLock(\*first);  std::lock\_guard<std::mutex> secondLock(\*second);  if (from->get\_balance() >= amount) {  from->set\_balance(from->get\_balance() - amount);  to->set\_balance(to->get\_balance() + amount);  return 0;  }  return -1;  }  void f(BankAccount \*ba1, BankAccount \*ba2) {  std::thread thr1(deposit, ba1, ba2, 100);  std::thread thr2(deposit, ba2, ba1, 100);  thr1.join();  thr2.join();  } |

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

| **Principles(s):** Reliable Resource Management  This principle ensures consistent system behavior and resource management in multi-threaded environments to prevent deadlocks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probably | Medium — Requires moderate effort to refactor existing locking mechanisms. | P4 - Important but not critical to address immediately. | L3 - Requires attention to prevent occasional disruptions in service. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | CONCURRENCY.LOCK.ORDER | Detects potential deadlocks by analyzing conflicting lock orders. |
| Coverity | 6.5 | DEADLOCK | Detects conditions leading to deadlocks to ensure safe lock acquisition. |
| Helix QAC | 2024.1 | C++1772, C++1773 | Checks for locking patterns that may lead to deadlocks. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CON53-a | Ensures locks are acquired in a consistent order. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Container Safety | STD-010-CPP | Guarantee that container indices and iterators are within the valid range  Proper management of container indices and iterators is essential to prevent errors such as out-of-bounds access, which can lead to security vulnerabilities, application crashes, or unexpected behavior. Making sure that operations on containers are performed within their valid boundaries strengthens program stability and security. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates potential out-of-bounds access due to insufficient checking of a vector's index bounds, risking negative indexing on systems where size\_type is unsigned. |
| #include <vector>  void insert\_in\_table(std::vector<int>& table, long long pos, int value) {  if (pos >= table.size()) { // Fails to prevent negative indexing  // Handle error  return;  }  table[pos] = value; // Potentially accesses out-of-bounds  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that all indices are checked against the vector's size, and it uses the at() method which provides built-in bounds checking and throws an exception if the index is out of range. |
| #include <vector>  void insert\_in\_table(std::vector<int>& table, std::size\_t pos, int value) noexcept(false) {  table.at(pos) = value; // Throws std::out\_of\_range if pos is out of bounds  } |

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

| **Principles(s):** Reliability and Security  This standard promotes reliability by preventing errors such as out-of-bounds access, which are critical for maintaining system security and stability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High - Requires thorough testing and code review. | P9 - Essential for system security and stability. | L2 - Significant impact on application reliability. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference | Detects overflows upon dereferencing, ensuring safe array and pointer use. |
| CodeSonar | 8.1p0 | LANG.MEM.BO, LANG.MEM.BU | Identifies buffer overruns and underruns to prevent access violations. |
| Klocwork | 2024.1 | ABV.GENERAL | General checks for array bounds violations to ensure memory safety. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR50-a | Verifies that container indices and iterators stay within valid bounds. |

### 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.

**Written Explanation**

To automate the enforcement of security standards at Green Pace, it's essential to integrate security automation tools across the DevSecOps pipeline depicted in the provided diagram. In the planning and design stages, tools such as static code analyzers can assess potential security threats early, while automated security testing tools during the build phase ensure continuous compliance and early detection of vulnerabilities.

During deployment, configuration management tools like Ansible can automate secure deployment processes. After deployment, continuous monitoring with tools such as Splunk ensures ongoing surveillance. This comprehensive integration of automated tools embeds security throughout the development lifecycle, enabling proactive threat detection and swift response to security incidents.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | High | Likely | Medium | P1 | L1 |
| STD-004-CPP | Low | Likely | Medium | P6 | L2 |
| STD-005-CPP | High | Likely | Medium | P2 | L1 |
| STD-006-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | Low | Probable | Medium | P4 | L3 |
| STD-010-CPP | High | Likely | High | P9 | L2 |

### 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 safeguards data stored on disk by converting it into a format that cannot be easily interpreted without a decryption key. This type of encryption is important for protecting sensitive information from unauthorized access, especially on storage mediums that can be physically stolen or accessed, such as hard drives, laptops, and external storage devices. The policy should be applied to all sensitive data, such as personally identifiable information, financial details, and intellectual property, ensuring it remains encrypted while in storage. This policy is applicable in environments with stringent data protection regulations and where data needs to be securely retained over long periods. |
| Encryption in flight | Encryption in flight focuses on securing data as it travels across networks to prevent interception or eavesdropping. This is achieved by using protocols such as HTTPS, FTPS, or TLS, which ensure that data transmitted between client devices and servers, or between servers, is encrypted. The policy applies whenever data is transmitted over untrusted networks, especially the internet, to protect against data breaches and ensure compliance with privacy laws. Implementing this policy is important for any data transfers involving sensitive information, including user credentials, payment details, or confidential business data. |
| Encryption in use | Encryption in use protects data that is actively being processed by applications. This type of encryption ensures data confidentiality and integrity, even when it is loaded into system memory. It is applicable in environments where data might be exposed to other applications or systems on the same platform, such as multi-tenant cloud environments or shared systems. The policy should be enforced when processing highly sensitive data, allowing it to remain encrypted even during computation, thereby mitigating risks associated with memory access violations and similar vulnerabilities. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication makes sure that users are who they claim to be by requiring credentials, such as passwords, biometrics, or security tokens, before accessing a system. This policy is essential for establishing secure access controls and maintaining user accountability. It should be applied at all points of user interaction with systems handling sensitive or restricted data. The policy dictates that authentication mechanisms must be robust, regularly updated, and compliant with industry standards to effectively prevent unauthorized access. |
| Authorization | Authorization determines the resources and operations that an authenticated user is permitted to access and perform. This policy should clearly define user permissions based on roles, responsibilities, and the principle of least privilege. It is essential for controlling access to sensitive data and resources within an organization, ensuring that users can only interact with data and actions necessary for their role. Authorization settings should be regularly reviewed and adjusted in response to changes in user roles or the security landscape. |
| Accounting | Accounting, or auditing, involves tracking and recording user activities to create an audit trail that can be used for monitoring, troubleshooting, and forensic analysis. This policy applies to all actions on sensitive systems and data, including user logins, database changes, and file access. It is essential for detecting and responding to security incidents, ensuring compliance with regulatory requirements, and providing a basis for post-event analysis. The policy should dictate the maintenance of detailed logs, regular review of access patterns, and the secure storage of audit trails to protect against unauthorized changes or deletions. |

**\***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 | 05/30/2024 | Completed milestone requirements | Mary Vuong |  |
| 1.2 | 06/16/2024 | Completed project requirements | Mary Vuong |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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