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



# Green Pace Secure Development Policy

# Created By: Tatiana Case

## 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 | An important security principle is validating input data, which is confirming that information entered into a system satisfies predetermined standards before processing. This lowers the possibility of malicious exploits and helps prevent a variety of attacks, including injection attacks, by ensuring that only expected and safe data is accepted. |
| 1. Heed Compiler Warnings | Compiler warnings are alerts that the compiler produces while compiling code that point out possible problems. Security depends on you taking note of and acting upon these alerts. Disregarding alerts may result in weaknesses and unexpected actions. Developers can identify potential security vulnerabilities and improve the overall robustness of the code by promptly addressing compiler warnings. |
| 1. Architect and Design for Security Policies | The procedures and guidelines that control how an organization approaches security are outlined in security policies. It is ensured that the architecture supports the organization's security objectives when systems are designed with security policies in mind. This covers things like data protection regulations, authentication procedures, and access controls. |
| 1. Keep It Simple | A fundamental security principle is simplicity. Vulnerabilities are frequently multiplied in complex systems. Developers can lessen the attack surface and make code easier to understand and secure by making designs and implementations as simple as possible. In general, easier to handle and less likely to introduce unintentional security risks are simple and uncomplicated solutions. |
| 1. Default Deny | According to the default deny principle, only resources that are specifically authorized should be accessible by default. This method makes sure that access is prohibited unless a specific permission is given. Both the chance of unauthorized access and the likelihood of missing security configurations are lessened. |
| 1. Adhere to the Principle of Least Privilege | The least amount of access or permissions required for users, processes, and systems to carry out their tasks is the focus of the least privilege principle. As a result, there is less chance of unexpected consequences from granting users too many privileges and less potential harm in the event of a security breach. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it is crucial when interacting with other systems to avoid compromising the receiving system with malicious content or code. Validating, purifying, and guaranteeing data integrity are all part of data sanitization, which guards against security flaws like data manipulation and injection attacks. |
| 1. Practice Defense in Depth | Implementing several tiers of security controls across a system or network is known as "defense in depth." This tactic makes sure that additional protection is provided by other layers even in the event of a breach in one. Organizations can improve their capacity to identify, stop, and mitigate different kinds of security threats by expanding their security measures. |
| 1. Use Effective Quality Assurance Techniques | Throughout the development process, quality assurance is essential for spotting and fixing security vulnerabilities. Code reviews, extensive testing, and other quality control methods assist in identifying and addressing vulnerabilities prior to their implementation in production. This proactive approach aids in the development of software that is more dependable and secure. |
| 1. Adopt a Secure Coding Standard | Best practices and guidelines for writing secure code are provided by a secure coding standard. Following such guidelines lessens the possibility of introducing common security vulnerabilities and helps to ensure consistency. It offers developers a guide to safe coding techniques, addressing things like input validation, handling errors, and encryption. |

### 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** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Avoid using superfluous qualifiers on reference types, such as data types, in order to preserve code readability, consistency, and clarity. Using the terms "volatile" or "constant" to qualify a reference is superfluous and may cause misunderstanding without offering any real advantages. These qualifiers complicate and make the code more difficult to read. |

| **Noncompliant Code** |
| --- |
| An integer (x) is referred to by a constant reference (const int& refX). There is no extra safety or benefit to using const with a reference to an int, so it is redundant. Without a good reason, it complicates the code, which could confuse readers. In this case, the const-qualified reference is superfluous and goes against the guideline. |
| int main(){  int x = 74;  const int& refX = x; //redundant use of a constant reference  cout << “Value of x:” << refX << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant code follows the guidelines by not using const to qualify a reference when it isn't necessary. Because the reference (int& refX) is non-const, redundancy is removed and the code is more readable. The reference doesn't have to be constant because it doesn't stop changes to the underlying variable |
| int main(){  int x = 74;  int& refX = x; //reference without using a constant qualifier  cout<< “Value of x:” << refX << endl;  return 0;  } |

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

| **Principles(s):**   * Keep It Simple: Extraneous qualifiers complicate the code and make it more difficult to read. A simple and uncluttered reference type adds to the codebase's overall security and simplicity. * Validate Input Data: Extraneous qualifiers on reference types are considered inputs to the code and keeping them in a consistent format helps improve the readability and overall quality of the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.17 | V201 Multiple Definitions | Provides an extra layer of protection against violations and finds problems with multiple definitions. |
| Checkmarx | 10.0 | C++ MultDef | A tool for static application security testing that finds security holes in the source code. It can be set up to identify and highlight situations in which reference types are overly qualified in order to comply with the coding standard. |
| Coverity | 2022.1 | DCL60-CPP Checker | Single definition rule enforcement and autoscanning. To guarantee that developers receive feedback during the build process, integrate with the CI/CD pipeline. |
| Axivion Bauhaus Suite | 7.0 | MISRA C++ | Concentrates on finding several declarations within the same scope. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002- CPP | The integrity of the data may be jeopardized and undefinable behavior may result from casting to an out-of-range enumeration value. To keep things consistent and avoid unforeseen problems, always make sure the value being cast is within the enumeration's valid range. |

| **Noncompliant Code** |
| --- |
| Without first confirming that the integer is within the permitted range of enumerator values specified by the Color enumeration, an attempt is made to cast an integer (value) to an enumeration type (color). Undefined behavior might occur if the value is not within the acceptable range (that is, it is not one of Red, Green, or Blue). Because range validation is absent, errors and unexpected behavior could occur in the code. |
| enum Color { Red, Green, Blue };  void setColor(int value) {  Color color = static\_cast<Color>(value);// Casting w/o range  // Input Validation  // Color Processing  } |

| **Compliant Code** |
| --- |
| Before attempting the cast, the compliant code verifies that the integer value (value) is within the acceptable range of the Color enumeration. An error-handling mechanism is triggered if the value falls outside of the acceptable range. By ensuring that the cast is only executed when the integer matches a legitimate Color enumerator, this improves the code's resilience and prevents undefined behavior. |
| enum Color { Red, Green, Blue };  void setColor(int value) {  if (value >= Red && value <= Blue)  {  // Validation check for the range  Color color = static\_cast<Color>(value); // Casting with range  // Input Validation  // Color Processing  }  else {  // Error handler for out-of-range value  }  } |

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

| **Principles(s):**   * Validate Input Data: Ensuring that only expected and secure data is used helps prevent potential security flaws and validates that the integer being cast falls within the valid range of the enumeration. * Adopt a Secure Coding Standard: This coding standard offers detailed instructions on how to prevent undefined behavior and data integrity issues caused by casting integers to values that are outside of the range of enumeration. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 10.8 | enum-cast | A checker made expressly to find problems with casting between integers and enumerations is offered by Clang Static Analyzer. By configuring this checker, you can help it automatically detect and mark instances in which casting to an out-of-range enumeration value happens. |
| PVS-Studio | 7.22 | V570 | One rule (V570) in the static analysis tool PVS-Studio is specifically made to detect problems arising from improper use of enumerations. |
| Coverity | 2023.9.0 | enum-cast | An instrument for static analysis with a checker built in to find problems with enum casting. By using this checker, you can make sure that the codebase is automatically searched for casting to values that are outside of the range of enumerations. |
| FindBugs | 3.0.1 | DMI\_BAD\_CAST | FindBugs' DMI\_BAD\_CAST detector is designed to find problems associated with casting. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Make sure there is enough room in the string storage for both the null terminator and character data. This procedure guarantees that strings are handled correctly in C++ and stops buffer overflows. |

| **Noncompliant Code** |
| --- |
| The "Hello" string is attempted to be copied into a buffer (str) of a predetermined length by the function processString. Nevertheless, the null terminator is not taken into consideration when calculating copyLength. There won't be room for the null terminator if the buffer's length is precisely the same as the length of "Hello," which could result in buffer overflows and undefinable behavior. |
| #include <cstring>  void processString(char\* str, size\_t length) {  // Not enough space for null terminator  size\_t copyLength = std::min(length, strlen("Hello"));  std::strncpy(str, "Hello", copyLength);  // Process  } |

| **Compliant Code** |
| --- |
| To solve the problem, the compliant code calculates copyLength and leaves room for the null terminator. To ensure that the string is correctly terminated, it also explicitly adds the null terminator. This keeps the buffer from overflowing and guarantees proper string handling by making sure it is big enough to hold the character data and the null terminator. |
| #include <cstring>  void processString(char\* str, size\_t length) {  // Make enough space for the null terminator  size\_t copyLength = std::min(length - 1, strlen("Hello"));    std::strncpy(str, "Hello", copyLength);  str[copyLength] = '\0'; // Ensure null terminator  // Process  } |

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

| **Principles(s):**   * Sanitize Data Sent to Other Systems: By making sure that strings are handled correctly and that a null terminator is present, security flaws like buffer overflows and possible data manipulation can be avoided when interacting with other systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 17.7 | V518 Checker | Designed to look for problems and security issues as well as identify different kinds of errors and vulnerabilities. |
| Coverity | 2022.1 | STR31-CPP | To guarantee ongoing monitoring, automatically check for infractions pertaining to inadequate storage for strings. |
| Parasoft | 2022.2 | STR31-e Check | A security testing tool's objectives were to decrease errors, boost productivity, and avert possible security breaches during the software development process. |
| TrustInSoft Analyzer | 1.38 | Mem\_Access Check | Aids in the discovery and removal of critical defects, compliance problems, and security vulnerabilities by developers. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | File streams are frequently opened by developers for input and output operations. When transferring between input and output operations in such a scenario, it is imperative to use positioning calls (such as seekg, seekp) to move the file stream cursor. This preserves the integrity of the file's read and write operations and avoids unforeseen consequences. Incorrect handling of untrusted data within SQL queries can lead to SQL injection vulnerabilities when input and output operations are alternated between without proper positioning. |

| **Noncompliant Code** |
| --- |
| On a file stream (file), the noncompliant code switches between input and output operations without making a positioning call in between. This may result in unanticipated outcomes like the output erasing previously written material, reading inaccurate data, or corrupting data. Unintended behavior may arise from forgetting to move the file pointer before converting between input and output. |
| #include <fstream>  #include <iostream>  void processFile(const std::string& filename, const std::string& userInput) {  std::fstream file(filename, std::ios::in | std::ios::out);  // Noncompliant: Alternating input and output w/o positioning  file << "User input: " << userInput << std::endl;  std::string userData;  file >> userData;  // Process userData  } |

| **Compliant Code** |
| --- |
| In order to ensure correct positioning before transitioning from output to input operations on the file stream, the compliant code uses seekp. Prior to performing input, this looks for the file pointer to the beginning (std::ios::beg), avoiding unforeseen outcomes and possible file content corruption. The file operations are well-defined and do not conflict with one another when they are repositioned properly. |
| #include <fstream>  #include <iostream>  void processFile(const std::string& filename, const std::string& userInput) {  std::fstream file(filename, std::ios::in | std::ios::out);  // Compliant: Repositioning before alternating  file << "User input: " << userInput << std::endl;  file.seekp(0, std::ios::beg); // Reposition to the beginning before input  std::string userData;  file >> userData;  // Process userData  } |

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

| **Principles(s):**   * Adhere to the Principle of Least Privilege- Following this guideline reduces the possible harm that could result from security breaches. In accordance with the least privilege principle, repositioning the file stream cursor prior to switching between input and output operations guarantees that each operation is clearly defined and prevents unforeseen consequences. * Sanitize Data Sent to Other Systems- When working with untrusted data, moving the file stream cursor can help avoid unintentionally handling user input during file operations, which lowers the chance of data corruption or manipulation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Occasional | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQLMap | 1.4.12 | Sqlmap-H Check | An automatic scanning tool that finds and takes advantage of SQL Injection Vulnerabilities and streamlines the detection process |
| FindBugs | 3.0.1 | SQL String Passed | A static analysis tool that finds common bugs and assists developers in detecting problems and maintaining code quality |
| Coverity | 2022.2 | SQLI Check | A static code analysis tool with the ability to detect security flaws and look for vulnerabilities early on |
| Parasof | 2022.2 | Security SQL Check | A tool that helps with SQL checks can guarantee that coding standards are followed and that infractions are dealt with. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | When allocating memory for objects or arrays whose size may not be known at compile time, C++ frequently uses dynamic memory allocation. In order to stop memory leaks and return resources to the system, proper deallocation is essential. This policy is applicable to all resources that are acquired through dynamic allocation, such as memory, file handles, network connections, and other resources. Deallocating dynamically allocated memory is necessary to prevent memory leaks, which occur when unreleased memory builds up over time and may result in the program running out of memory. |

| **Noncompliant Code** |
| --- |
| There is a memory leak when dynamically allocated memory is not deallocated using delete[]. Memory exhaustion may occur if dynamically allocated memory is not released, leading to inefficient memory usage. |
| #include <iostream>  Void allocateMemory() {  int dynamicArray = new int[20];  // Process dynamicArray  // Deallocation Missing  } |

| **Compliant Code** |
| --- |
| Using delete, the compliant code appropriately deallocates the dynamically allocated memory[]. By doing this, memory leaks are avoided because it guarantees that the memory is returned to the system. |
| #include <iostream>  Void allocateMemory() {  int dynamicArray = new int[20];  // Process dynamicArray  Delete[] dynamicArray; // Proper Deallocation  } |

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

| **Principles(s):**   * Adopt a Secure Coding Standard- Making sure that dynamically allocated resources are properly dealt with in accordance with the secure coding principle. It places a strong emphasis on adhering to accepted guidelines in order to guard against common flaws like memory leaks and to preserve the general security and stability of the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | Return Value Check | An open-source program that includes a check for potential memory access errors and memory invalidation calls |
| TrustInSoft Analyzer | 1.38 | Location Value Check | Tool that supports rigorous analysis of issues pertaining to memory access and uses formal methods to mathematically verify code |
| Coverity | 2022.1 | MEM50-CPP Checker | In order to guarantee continuous monitoring and feedback to be relayed, auto scan violations pertaining to accessing freed memory and integrating this tool into the pipeline are required. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | To verify the value of a constant expression at compile time, use a static assertion (static\_assert). This eliminates the runtime overhead associated with dynamic assertions and guarantees that the condition is validated during the compilation process, providing early error detection. |

| **Noncompliant Code** |
| --- |
| A constant expression (MAX\_VALUE) is checked using a runtime assertion (assert). However, since the condition is known at compile time, assertions are typically used for runtime checks. It adds needless overhead to the program's execution to use a runtime assertion, and any potential problems with the constant expression won't be discovered until runtime. |
| #include <cassert>  constexpr int MAX\_VALUE = 200;  void processData(int value) {  assert(value <= MAX\_VALUE); // Noncompliant: Runtime assertion for a  // constant expression  } |

| **Compliant Code** |
| --- |
| The constant expression (MAX\_VALUE) is validated at compile time using a static assertion (static\_assert). During the compilation process, the compiler verifies the condition; if it is not met, an error message is generated along with the compilation error. This eliminates runtime overhead and guarantees early detection of problems with the constant expression. |
| #include <cassert>  constexpr int MAX\_VALUE = 200;  void processData(int value) {  static\_assert(value <= MAX\_VALUE, "Value exceeds MAX\_VALUE");  // Compliant: Static assertion for a constant expression  } |

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

| **Principles(s):**   * Use Effective Quality Assurance Techniques- When used properly, assertions help improve the quality and dependability of the code by making it possible to identify errors early on in the compilation process, especially in constant expressions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 17.7 | Assertion Checker | Static code analysis tool that ensures that assertions are implemented correctly by detecting their usage and checking for a variety of problems. |
| Coverity | 2022.1 | DCL53 CPP Check | The tool will help by automatically identifying and stopping unclear or confusing code while preserving clarity. |
| Clang Static Analyzer | 14.0 | DCL53-e Check | This tool, which is a component of the clang compiler, can help identify assertion statement issues during static analysis. |
| FindBugs | 3.0.1 | ClrCdCheck | This will assist in locating issues with assertion statements and offer recommendations for enhancements. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | The ability to handle exceptions is essential to writing dependable software. Ignoring or improperly handling exceptions can cause unpredictable behavior, complicate debugging, and possibly cause runtime problems. |

| **Noncompliant Code** |
| --- |
| An exception may be thrown by the riskyOperation() function, but the catch block within the function catches the exception without processing or throwing it again. After that, the exception-related problems are not addressed and the main function continues. This kind of ignoring exceptions can result in ill-defined behavior and make it difficult to recognize and resolve issues. |
| #include <iostream>  void riskyOperation() {  try {  // Risky operation that may throw an exception  throw std::runtime\_error("An exception occurred");  } catch (std::exception& e) {  // Exception caught, but no handling or rethrowing  }  }  int main() {  riskyOperation();  // Code continues after riskyOperation, issues not addressed  return 0;  } |

| **Compliant Code** |
| --- |
| The function riskyOperation() has been altered to appropriately manage the exception. The function's catch block optionally rethrows the exception and prints an error message to std::cerr. To handle exceptions at a higher level, an extra try-catch block is added to the main() function. This keeps the program from running with possible problems unfixed and enables appropriate logging or handling of exceptions. |
| #include <iostream>  void riskyOperation() {  try {  // Risky operation that may throw an exception  throw std::runtime\_error("An exception occurred");  } catch (std::exception& e) {  // Proper handling or rethrowing of the exception  std::cerr << "Exception caught: " << e.what() << std::endl;  // Optionally rethrow exception  throw;  }  }  int main() {  try {  riskyOperation();  // Code continues after riskyOperation, issues addressed  } catch (std::exception& e) {  Std::cer << “Main function exception caught: “ << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard- Makes certain that exceptions are handled properly to avoid unforeseen runtime problems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tool Suite | 9.5 | ERR51-CPP Cert. | The all-inclusive tool LDRA is made for developing critical software and has the capacity to focus on adhering to security and safety regulations. |
| Parasoft | 2022.2 | ERR51-a | All-inclusive quality assurance tool that can perform static code analysis, improve overall reliability, and improve quality and defect identification. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| OOP | STD-008-CPP | The fundamentals of appropriate memory management and OOP techniques form the foundation of this standard. It highlights how, in order to guarantee accurate and thorough resource cleanup when working with polymorphic objects, virtual destructors must be used. By doing this, undefined behaviors, memory leaks, and bugs are avoided. |

| **Noncompliant Code** |
| --- |
| A pointer of type Base is assigned to a polymorphic object of type Derived. The destructor of the base class (Base), not the destructor of the derived class (Derived), is called when delete is called on the polymorphic object. As a result, there is insufficient cleanup and erratic behavior. If a polymorphic object is deleted without a virtual destructor, the program may behave incorrectly and resources may leak. |
| #include <iostream>  class Base {  public:  Base() {  std::cout << "Base constructor" << std::endl;  }  ~Base() {  std::cout << "Base destructor" << std::endl;  }  };  class Derived : public Base {  public:  Derived() {  std::cout << "Derived constructor" << std::endl;  }  ~Derived() {  std::cout << "Derived destructor" << std::endl;  }  };  int main() {  Base\* polymorphicObject = new Derived();  delete polymorphicObject; // Noncompliant: Deleting a polymorphic  // object without a virtual destructor  return 0;  } |

| **Compliant Code** |
| --- |
| One virtual destructor is included in the Base class. This guarantees that the appropriate destructor of the derived class (Derived) is called in tandem with delete when it is called on a pointer to the base class. In order to ensure correct resource cleanup in the derived class, using a virtual destructor in the base class is essential to polymorphic object deletion. |
| #include <iostream>  class Base {  public:  Base() {  std::cout << "Base constructor" << std::endl;  }  ~Base() {  std::cout << "Base destructor" << std::endl;  }  };  class Derived : public Base {  public:  Derived() {  std::cout << "Derived constructor" << std::endl;  }  ~Derived() {  std::cout << "Derived destructor" << std::endl;  }  };  int main() {  Base\* polymorphicObject = new Derived();  delete polymorphicObject; // Compliant: Deleting a polymorphic  // object with a virtual destructor  return 0;  } |

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

| **Principles(s):**   * Use Effective Quality Assurance Techniques- Developers help to create more dependable and safe software by making sure that polymorphic objects are handled carefully, especially in terms of appropriate memory management and the usage of virtual destructors. Using efficient quality assurance procedures improves the overall security of the codebase by identifying memory management and object destruction vulnerabilities before they appear in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.29 |  | Custom rules and checks can be created with PVS-Studio. The tool's adaptability allows it to be tailored to meet specific coding standards, such as the need for virtual destructors when working with polymorphic objects. |
| Coverity | 2023.9.0 | VirtualDestructor | The use of virtual destructors in polymorphic objects is the focus of the VirtualDestructor checker. |
| Clang Static Analyzer | 19.0.0 | VirtualCallChecker | The VirtualCallChecker finds problems with destructors and virtual function calls. |
| CppCheck | 2.13 | PolymorphicDelete | Polymorphic objects that are deleted without a virtual destructor are detected by the PolymorphicDelete checker. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | STD-009-CPP | It is essential to reference container elements using valid references, pointers, and iterators in order to avoid undefined behaviors, bounds checks, memory safety issues, and iterator invalidation. This guarantees that iterators, references, and pointers used to access elements inside a container are legitimate and stay inside the container's boundaries. |

| **Noncompliant Code** |
| --- |
| An attempt is made to use an iterator (invalidIterator) to access an element that is outside of the container's valid range (numbers). Because it involves accessing memory outside of the vector's boundaries, this results in undefinable behavior. Such actions may lead to program crashes, corrupted data, and erratic states. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> numbers = {1, 2, 3, 4, 5};  // Noncompliant: Invalid iterator  std::vector<int>::iterator invalidIterator = numbers.begin() + 10;  \*invalidIterator = 62; // Undefined behavior, accessing beyond  //container bounds  return 0;  } |

| **Compliant Code** |
| --- |
| The code that complies shows how to use a valid iterator (validIterator) that falls inside the container's allowed range (numbers). Before dereferencing and altering the value, a check is made to make sure the iterator is not at or beyond the end() position. By using this method, undefined behavior related to accessing elements outside of the container's boundaries is prevented. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> numbers = {1, 2, 3, 4, 5};  // Compliant: Valid iterator  std::vector::iterator validIterator = numbers.begin() + 2;  if (validIterator != numbers.end()) {  \*validIterator = 62; // Valid operation within container bounds  } else {  // Handle case where iterator is located or beyond  std::cerr << "Invalid iterator position." << std::endl;  }  return 0;  } |

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

| **Principles(s):**   * • Validate Input Data- Places a strong emphasis on verifying and making sure that data entered into a system—in this case, the container—meets requirements before processing. To ensure appropriate access to container elements and prevent undefinable behaviors, valid references, pointers, and iterators must be used when validating input data in the context of containers. This lowers the possibility of vulnerabilities and unintended issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.6.0 | ContainerErrors | The purpose of the ContainerErrors checker is to identify errors pertaining to containers, such as problems with iterators and container bounds. |
| CppCheck | 2.7 | ContainerBounds | Container bound problems in C++ code are detected by the ContainerBounds checker. It assists in locating possible issues with container accesses and iterators. |
| Clang Static Analyzer | 19.0.0 | llvm-undefined-behavior | Problems with undefined behavior in C++ code can be found with the llvm-undefined-behavior checker. |
| PVS Studio | 7.29 | IteratorValidity | Invalid iterators in C++ code are detected by the IteratorValidity checker. It assists in ensuring that iterators are used appropriately to avoid memory safety issues, bounds checks, and undefined behavior. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | STD-010-CPP | Code quality is improved, buffer overflows are avoided, memory settings are maintained, undefined behaviors are prevented, and library functions are guaranteed not to overflow. Ensuring the library operates is crucial for security because buffer overflows can result in memory corruption and undefinable behaviors. |

| **Noncompliant Code** |
| --- |
| To access an element at an index (7) outside of the vector's valid range, use the std::vector function at(). If the index is outside of bounds, the at() function checks bounds and throws std::out\_of\_range. But in this instance, the exception goes unnoticed, which could result in undefinable behavior and even program termination. It is not guaranteed by the noncompliant code that library functions won't overflow. |
| #include <iostream>  #include <vector>  int main() {  const size\_t size = 6;  std::vector<int> numbers(size);  // Noncompliant: Using library function without checking bounds  int value = numbers.at(7); // Throws std::out\_of\_range, but not  // caught  return 0;  } |

| **Compliant Code** |
| --- |
| If the index is out of bounds, the at() function throws an std::out\_of\_range exception, which the compliant code uses the try-catch mechanism to catch. The code stops undefined behavior and gives you a chance to react to the error condition by handling the exception. This compliant method incorporates appropriate bounds checking and error handling to ensure that library functions do not overflow. |
| #include <iostream>  #include <vector>  int main() {  const size\_t size = 3;  std::vector<int> numbers(size);  // Compliant: Using library function with bounds checking  try {  int value = numbers.at(2); // Valid index within the vector's  // range  } catch (const std::out\_of\_range& e) {  // Handle the out-of-range exception  std::cerr << "Out of range exception: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):**   * Principle of Least Privilege- Highlights how crucial it is to stop buffer overflows and make sure library functions don't run out of memory. Applying bounds checking and managing exceptions suitably are two ways the code adheres to the idea of granting the least amount of access required to complete the task. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS Studio | 7.29 | V612 | The V612 checker finds problems with out-of-bounds access to array elements. |
| Coverity | 2023.6.0 | BUFFER\_SIZE\_CHECK | The BUFFER\_SIZE\_CHECK checker can guarantee the safe use of library functions by spotting possible buffer overflows. |
| Cppcheck | 2.7 | ArrayIndexOutOfBounds | The purpose of the ArrayIndexOutOfBounds checker is to identify problems with accessing array elements outside of permitted boundaries. |
| Clang Static Analyzer | 19.0.0 | V612 | The V612 checker finds problems with out-of-bounds access to array elements. |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

[Insert your written explanations here.]

### 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 | Medium | Likely | Low | Medium | 2 |
| STD-002-CPP | High | Likely | Medium | High | 2 |
| STD-003-CPP | High | Likely | Medium | High | 2 |
| STD-004-CPP | High | Occasional | Medium | Medium | 3 |
| STD-005-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-006-CPP | Medium | Unlikely | Low | Low | 3 |
| STD-007-CPP | Low | Occasional | Medium | Low | 3 |
| STD-008-CPP | High | Medium | Low | High | 4 |
| STD-009-CPP | High | Medium | Medium | High | 4 |
| STD-010-CPP | Medium | Medium | Medium | Medium | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Safeguarding data while it's kept in file systems, databases, or other storage devices is known as encryption at rest. This policy covers all private and sensitive information kept on the systems of the company. |
| Encryption in flight | Encryption in flight provides data security during network transmission. In order to safeguard data during transmission from interception and unauthorized access, this policy is crucial. |
| Encryption in use | When encryption is used, data is safeguarded during processing or application use. In order to protect sensitive data while it is being processed actively, this policy is essential. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying the identity of users and systems gaining access to the organization's resources is the process of authentication. This policy makes sure that sensitive information can only be accessed by authorized parties. |
| Authorization | The degree of access that authenticated users are granted is determined by authorization. Users can only access the resources required for their roles, thanks to this policy. |
| Accounting | Accounting entails keeping tabs on user activity within the systems used by the company. This policy is essential for keeping track of user actions in an audit trail. |

**\***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 log

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 06/16/2024 | Completed Template | Tatiana Case |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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