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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Always check and verify the data that is coming into your system. This helps stop harmful or unexpected data from causing problems or being used to attack an application. For example, input from users or other systems could be checked for things like format, length, and content to make sure it is safe, and it is what the program expects. |
| 1. Heed Compiler Warnings | Always pay attention to warnings from the compiler when you are building code. These warnings are meant to point out problems that lead to bugs or other security issues. Fixing them as early as possible not only makes the software more reliable but it also makes it secure before it is available to the user. |
| 1. Architect and Design for Security Policies | Security is always a primary focus for software designing from the very beginning. This means we need to think about how data is protected, how users are verified, and how each action is tracked. Planning for security early on can help avoid expensive fixes later. |
| 1. Keep It Simple | Simple code is much easier to understand, test, and keep secure. When code gets more complicated, this can lead to hidden problems or mistakes that could severely affect the software. Writing clear and easy to understand code reduces the chance of unintended security flaws. |
| 1. Default Deny | When designing systems, it is always good to think of everything as denied unless it is specifically allowed. Using this method, we can make sure only trusted users and actions are allowed. This can help prevent accidental access for unauthorized users or systems and can provide an extra layer of protection. |
| 1. Adhere to the Principle of Least Privilege | Make sure users and systems have the minimum amount of access they need to complete their job. This can help limit the damage they can do if something goes wrong or if an account has been compromised. A good example is popular database setups where a user who only needs to read data should not be able to delete it. |
| 1. Sanitize Data Sent to Other Systems | Before we send data to a different system such as a database, it is important to clean it to remove anything problematic or unexpected. This helps to prevent attacks such as SQL injections or XXS (cross-site scripting), where bad data is used to confuse or break the system. |
| 1. Practice Defense in Depth | It is always important to use multiple layers of security so that if one of the layers isn’t enough, others are in place to protect the system. For example, even if a password is leaked, a second layer can be in place such as two-factor authentication to keep the account safe. |
| 1. Use Effective Quality Assurance Techniques | You always want to test software as often as you can using tools that are capable of checking for bugs and potential security issues. This can include things such as code reviews, automated testing, and testing it yourself. Catching problems as soon as possible makes the software more reliable and secure. |
| 1. Adopt a Secure Coding Standard | Following the secure coding rules that everyone on the team would agree to can help put everyone on the same page. Following these rules can help avoid commonly made mistakes and make sure the code is consistent and safe. It also helps new members see how to write the secure code as soon as they are hired. |

### 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 | Using the best data types for the variables. Using the best data types prevents unexpected things from happening, memory problems, and loss of data. It improves the code and makes sure the variable is large enough to store everything without causing overflow or truncation. |

| **Noncompliant Code** |
| --- |
| This code is one of the examples that uses a char type for a number based calculation, which leads to problems with loss of data or something unexpected happening |
| char result;  result = 150 + 120; // Using something like this could create overflow. |

| **Compliant Code** |
| --- |
| This example uses int instead of char, which is much better for holding the values required. |
| int result;  result = 150 + 120; // This keeps overflow from happening. |

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

| **Principles(s):**  Principle 4: Keep it Simple – Picking the right data type improves code and makes it more clear and reduces unexpected behavior.  Principle 1: Validate Input Data – Making sure variables can store the expected data types can help maintain safe and valid operations in the entire program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | typeOverflow | Finds wrong type usage that could lead to overflow problems. |
| Clang-Tidy | 17.0.1 | cppcoreguidelines-pro-type-implict-conversion | Focuses on dangerous or implicit conversions between the types. |
| SonarQube | 9.9 LTS | cpp:S5246 | Finds situations where data type could cause overflow regarding sizing or conversions. |
| PVS-Studio | 7.26 | V656 | This can warn you about possible data loss during implicit type conversions, especially when it sees small to large integer types. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | This makes sure there is a good range of data values. Using data ranges outside of their expected range can cause a lot of errors, overflow, underflow, or possibly even security problems. Making sure that values are within their limitations can help protect the software from unexpected problems and increase the overall health of the code. |

| **Noncompliant Code** |
| --- |
| This code wouldn’t check the user’s input and see if it is within range before executing the operation, which can cause the code to do expected actions or possible problems with memory. |
| #include <iostream>  void setScore(int score) {  int scores[100];  scores[score] = 1; // This does not allow for a check on the score values.  }  int main() {  int userScore;  std::cin >> userScore;  setScore(userScore); // Could cause out-of-bounds access to threats.  return 0;  } |

| **Compliant Code** |
| --- |
| These changes would allow us to validate the user’s input to make sure the value is within the valid ranges before using it, preventing out-of-bounds access. |
| #include <iostream>  void setScore(int score) {  int scores[100];  if (score >= && score < 100) {  scores[score] = 1; // Allows for safe access.  } else {  Std::cerr << “Out of range.” << std::endl;  }  }  int main() {  int userScore;  std::cin >> userScore;  setScore(userScore); // Validating the input before it is used.  return 0;  } |

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

| **Principles(s):**  Principle 1: Validate Input Data – It is always good to check that values are within the valid expected range to keep unexpected behaviors from happening like buffer overflows or underflows.  Principle 5: Default Deny – If a value is not within the expected range, access would be denied, making sure the system stays protected against the wrong inputs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | outOfBounds | Detects possible out-of-bounds array access problems. |
| Clang-Tidy | 17.0.1 | clang-analyzer-core.CallAndMessage | Flags unvalidated inputs used in array indexing. |
| SonarQube | 9.9 LTS | cpp:S3518 | Highlights missing boundary checks that could cause memory access problems. |
| Coverity | 2024.03 | ARRAY\_INDEX\_OUT\_OF\_BOUNDS | Finds any unsafe array indexing and missing input validation in programs such as C/C++. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | This would make sure the string operations are null-terminated, and the bounds are checked. Using C strings improperly can cause things like buffer overflows, memory problems, and application crashes. Making sure that the strings are null-terminated, and the operation bounds are checked, stopping any vulnerabilities and making the string data is handled correctly |

| **Noncompliant Code** |
| --- |
| This would use strcopy() without checking the length of the input, making it possible for buffer overflow if userInput is longer than the final buffer. |
| #include <cstring>  void copyUserInput(cont char\* userInput) {  char buffer[10];  strcopy(buffer, userInput); // This sets no bounds.  } |

| **Compliant Code** |
| --- |
| Using this version uses strcopy() and makes sure the destination is null-terminated, avoiding the overflow and making sure there is reliable and safe string copying. |
| #include <cstring>  void copyUserInput(const char\* userInput) {  char buffer[10];  strcopy(buffer, userInput, sizeof(buffer) -1); // This allows for copy with bounds set.  buffer[9] = ‘\0’; // This allows for null-termination.  } |

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

| **Principles(s):**  Principle 7: Sanitize Data Sent to Other Systems – Handling string properly can make sure that user input will not overflow into memory accidentally.  Principle 9: Use Effective Quality Assurance Techniques – Checking the bounds and null-termination of strings can help catch the vulnerabilities early when using secure coding and proper validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | bufferAccessOutOfBounds | It can see when buffer limits are over the limits when using string operations. |
| Clang-Tidy | 17.0.1 | Clang-analyzer-cplus.NewDelete | Finds unsafe C++ memory and string operations, including missing null terminations. |
| SonarQube | 9.9 LTS | cpp:S3510 | Highlights possibly harmful buffer manipulations that could lead to overflow or missing string terminators. |
| Coverity | 2024.03 | BUFFER\_SIZE | Is able to find string copying issues, improper buffer size issues, and missing bull termination in C/C++ software. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Preventing SQL injections by setting limits. SQL injections happen when there is an untrusted input that is in the SQL code, allowing malicious entities to query it and cause problems. Using queries with limits makes sure that the user input acts as data, not as code, keeping the hacker from getting unauthorized access or taking control. |

| **Noncompliant Code** |
| --- |
| This code takes the user unput and puts it directly into the SQL query, allowing the attacker to cause an SQL injection and affect the database. |
| #include <string>  #include <iostream>  void getUserData(const std::string& username) {  std::string query = “SELECT \* FROM user WHERE name = ‘ “ + username + “ ‘ “;  // This executes a query and is vulnerable to SQL injections.  } |

| **Compliant Code** |
| --- |
| This sets limits to the query including something like SQlite which we used in our assignment, making sure that the input is safe from queries, preventing SQL injections. |
| #include <mysql/mysql.h>  void getUserData(MYSQL\* conn, const std::string& username) {  MYSQL\_STMT\* stmt = mysql\_sttmt\_init(conn);  const char\* query = “SELECT \* FROM user WHERE name = ?”;  mysql\_stmt\_prepare(stmt, query, strlen(query));  MYSQL\_BIND bind[1] = {};  bind[0].buffer\_type = MYSQL\_TYPE\_STRING;  bind[0].buffer = (char\*)username.c\_str();  bind[0].buffer\_length = username.length();  mysql\_stmt\_bind\_param(stmt, bind);  mysql\_stmt\_execute(stmt);  mysql\_stmt\_close(stmt);  } |

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

| **Principles(s):**  Principle 7: Sanitize Data Sent to Other Systems – Protects the database by making sure that user input is treated like data, not code.  Principle 3: Architect and Design for Security Policies – Makes sure there is a focus on safe design practices by using prepared statements to avoid any vulnerabilities during the system architecture and coding phases. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 LTS | cpp:S3649 | Finds vulnerabilities where untrusted data may be used to build things such as SQL queries. |
| Coverity | 2024.03 | TAINTED\_STRING | Detects cases where there is unsanitized user input passed onto SQL queries, which could lead to injection risks. |
| CodeQL | 2.16.3 | cpp/sql-injection | Static analysis query that can find SQL injection vulnerabilities in C++ code. |
| Fortify Static Code Analyzer | 24.1 | SQL Injection | Focuses on dataflow and highlights places where user input might affect SQL execution paths. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | It’s good to always have free dynamically allocated memory and you want to try and avoid dangling pointers. When you do not release this memory, it can lead to leaks, while using memory after it is freed or dangling pointers this can cause unexpected things to happen or create vulnerabilities hackers can take advantage of. You want to release memory safely and set pointers to nullptr after it is used and this with help with both the stability and security. |

| **Noncompliant Code** |
| --- |
| This code will show when the code allocates the memory but does not free it, causing memory leaks. Over time, this can severely affect the speed of the system of even cause it to crash. |
| #include <iostream>  void createData() {  int\* data = new int[100];  // No function here to delete and can cause the memory to leak.  std::cout << “Data has been created” << std::endl;  } |

| **Compliant Code** |
| --- |
| This version releases the memory and the pointer is set to nullptr, allowing us to avoid leaks and reducing the risk of dangling pointers. |
| #include <iostream>  void createData() {  int\* data = new int[100];  // This is where the data would be used.  delete[] data;  data = nullptr;  std::cout << “Data cleaned.” << std::endl;  } |

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

| **Principles(s):**  Principle 8: Practice Defense in Depth – Even if some memory leaks happen, setting pointers to nullptr adds an extra layer of safety against dangling pointer usage.  Principle 9: Use Effective Quality Assurance Techniques – Good memory management is encouraged through careful coding, inspections, and runtime analysis to stop leaks and crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | memleak | Finds memory leaks where allocated memory would not be properly released. |
| Clang-Tidy | 17.0.1 | clang-analyzer-cplusplus.NewDeleteLeaks | Flags dynamically located memory that would not be properly deallocated. |
| Valgrind | 3.22.0 | Memcheck | Runtime check tool that shows memory leaks, improper frees, and use-after-free errors that appear. |
| Coverity | 2024.03 | RESOURCE\_LEAK | Finds dynamically allocated memory that is lost without being released the proper way. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | You do not want to have assertion when you are trying to validate the more important runtime conditions. Assertions are great during the development process to catch bugs as early as possible, but they can be turned off when the program is sent to production. Using assertions to check for important conditions at runtime can cause problems with security and create vulnerabilities if the checks are completely removed. You always want to use good error handling for the more critical conditions. |

| **Noncompliant Code** |
| --- |
| To show the error, this code will use an assert to validate user input. If assertions are disabled in production, the check will be skipped and can cause some unexpected things to happen. |
| #include <cassert>  #include <iostream>  void processInput(int value) {  assert(value != 0); // This is unsafe because the assertion could be disabled.  Std::cout << 100 / value << std::endl;  } |

| **Compliant Code** |
| --- |
| This code includes error handling instead to validate the input, making sure the check is always done, even during the production build. |
| #include <iostream>  #include <stdexcept>  void processInput(int value) {  if (value == 0) {  throw std::invalid\_arguement(“Value can’t be a zero”);  }  std::cout << 100 / value << std::endl;  } |

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

| **Principles(s):**  Principle 4: Keep it Simple – Using a good error handling method instead of depending on assertions makes sure there is a more predictable and consistent behavior.  Principle 9: Use Effective Quality Assurance Techniques – Early testing should use assertion, but productions must keep to secure runtime check through efficient error handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | missingErrorHandling | Gives you a warning about missing error-handling when depending on assertions. |
| Clang-Tidy | 17.0.1 | bugprone-assert-side-effect | Identifies improper or risky use of assert, especially if it is used for important runtime checks. |
| SonarQube | 9.9 LTS | cpp:S3519 | Finds risky use of assertions and recommends good use of organized exception/error handling instead. |
| Coverity | 2024.03 | ASSERT\_SIDE\_EFFECT | Identifies cases where assert statements are improperly used in place of proper runtime checks. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | You can catch exceptions by reference and handle properly. Catching exceptions by value can cause things such as object slicing and create problems with performance. Catching by reference instead saves the original exception type and keeps it from making unneeded copies. When exception handling is done right it can also make sure the program stays safe and consistent. |

| **Noncompliant Code** |
| --- |
| The code catches the exception by value, which causes slicing if the exception comes from somewhere else like a base class. It can also does not give us any useful error handling. |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  throw std::runtime\_error(“There was an error.”);  }  int main() {  try {  riskyOperation();  } catch (std::exception e) { // Catching it by value instead.  std::cout << e.what() << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| Catches the exception by reference to save the full object information and avoid any slicing. |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  throw std::runtime\_error(“There was an error.”);  }  int main() {  try {  riskyOperation();  } catch (const std::exception& e) {  Std::cout << e.what() << std::endl;  }  return 0:  } |

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

| **Principles(s):**  Principle 4: Keep It Simple – Good handling makes sure that runtime error processing is clear, predictable, and safe.  Principle 8: Practice Defense in Depth – Safe exception management keeps subtle runtime errors from getting worse and turning into security risks or unstable states. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | catchByValue | Catches exceptions caught by value instead of by a reference. |
| Cland-Tide | 17.0.1 | bugprone-exception-escape | Finds issues in exception handling, including wrong catch types. |
| SonarQube | 9.9 LTS | Cpp:S3656 | Makes sure that exceptions are caught by reference instead of value to prevent slicing. |
| PVS-Studio | 7.26 | V568 | Identifies incorrect exception catching issues that could cause data slicing or a loss of exception data. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Initialization | STD-008-CPP | You should initialize all variables before you use them. Using variables before they are properly initialized can cause unexpected behaviors, weird outputs, and major security problems. Initializing variables makes sure that the program starts with good data and avoids users from getting improper access or causing crashes. |

| **Noncompliant Code** |
| --- |
| This uses a variable without setting a value, causing unexpected behavior. |
| #include <iostream>  void printValue() {  int number; // Does not have an initialized value.  std::cout << number << std::endl; // Unexpected things will happen.  } |

| **Compliant Code** |
| --- |
| This code initializing the value before it is used, making everything much more predictable and safe for the program. |
| #include <iostream>  void printValue() {  int number = 0; // Shows the initialization.  std::cout << number << std::endl;  } |

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

| **Principles(s):**  Principle 1: Validate Input Data – Properly initialized variables make sure that no undefined or invalid data is processed.  Principle 4: Keep It Simple – Makes sure there is predictable and clean program behavior by starting all variables with valid values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | uninitvar | Flags things such as variables that are being used without being initialized first. |
| Clang-Tidy | 17.0.1 | clang-analyzer-core.uninitialized | Finds when variables are used before they are given a good initial value. |
| SonarQube | 9.9 LTS | cpp:S6000 | Identifies possible problems from uninitialized memory or variables. |
| PVS-Studio | 7.26 | V519 | Warns you about possible uninitialized variables being used in code execution paths. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Control Flow | STD-009-CPP | You want to avoid using goto for control flow. Using goto can make code difficult to understand, do maintenance on, and debug. This can also cause problems by skipping over important code or confusing some of the control structures implemented. Structured control flow like loops and conditionals would be used instead to provide a clearer and safer method. |

| **Noncompliant Code** |
| --- |
| Shows a commonly used baseline when goto used to skip parts of a function, making it difficult to follow along and maintain. |
| #include <iostream>  void example() {  int x = 0;  goto skip;  x = 42; // This would be skipped.  skip:  std::cout << x << std::endl;  } |

| **Compliant Code** |
| --- |
| This version of the code uses a conditional statement instead of a goto, improving the overall readability and build. |
| #include <iostream>  void example(bool condition) {  int x = 0;  if (!condition) {  x = 42;  }  std::cout << x << std::endl;  } |

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

| **Principles(s):**  Principle 4: Keep It Simple – Replacing goto with an organized control flow that improves code readability, reliability, and maintenance.  Principle 9: Use Effective Quality Assurance Techniques – Avoiding goto stops certain logical errors that can make debugging difficult and affect the verification efforts. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | gotoStatement | Finds the usage of goto statements and shows the review. |
| Clang-Tidy | 17.0.1 | readability-avoid-goto | Helps recommend the use of goto instead of a designed control flow. |
| SonarQube | 9.9 LTS | cpp:S3358 | Detects the existence of goto and recommends the use of designed loops and conditions. |
| PVS-Studio | 7.26 | V773 | Identifies the wrong use of goto statements that lead to skipped variable initialization or skipped logic statements. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | STD-010-CPP | It is always important to close file handles properly. When you do not close file handles properly, it can lead to leaks and file corruption. It is important to make sure that all opened files are closed in the entire execution path, including the error handling path to keep your file integrity and maintain system stability. |

| **Noncompliant Code** |
| --- |
| This code opens a file but never closes it, which could hurt your available resources or damage the file itself through corruption. |
| #include <fstream>  void writeToFile() {  std::ofstream file(“file.txt”);  file << “Hello!”; // This file was never closed.  } |

| **Compliant Code** |
| --- |
| You can use resource acquisition and use the file in a scope. The destructor would automatically close the file when it leaves the scope. |
| #include <fstream>  void writeToFile() {  std::ofstream file(“file.txt”);  if (file.is\_open()) {  file << “Hello!”;  } // This allows the file to close automatically when it is out of scope.  } |

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

| **Principles(s):**  Principle 8: Practice Defense in Depth – Makes sure that even if something like a write fails, file resources would still be released, and the system would remain stable.  Principle 9: Use Effective Quality Assurance Techniques – Proper file management supports better testing, monitoring of the code, and less resource leaks regarding audits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | resourceLeak | Find possible resource leaks where file handles are opened but not closed the right way. |
| Clang-Tidy | 17.0.1 | clang-analyzer-cplusplus.Move | Looks over the proper use and closure of resource-managing object like file streams. |
| SonarQube | 9.9 LTS | cpp:S2095 | Finds cases where resources such as file handling are not handled properly such as being closed or released. |
| Coverity | 2024.03 | RESOURCE\_LEAK | Finds file descriptors or file streams that are opened and not safely closed in all paths. |

### 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.

Green Pace uses security automation across the whole DevSecOps system, using security at each phase of software development and operations. For pre-production, automation would focus on secure design practices, secure builds using things such as trusted repositories, and constant verification through vulnerability scanning and compliance testing. Tools including Cppcheck, Clang-Tidy, SonarQube, and Coverity would be integrated directly into the build and verify stages to catch any security problems before the code is released,

As the applications would move into production, automation continues with the setting of deployment systems, SIEM tools, intrusion detection systems (IDS), and Higha real-time even monitoring system to find security problems. Automated responses would be invaluable to find threats, such as service rollbacks or telling you about escalations, these would be used to minimize downtime and exposure. Automation makes sure that Green Pace can maintain strong security, reducing the chance of human error and overall improving security, fully supporting the ideals of defense-in-depth.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | High | 4 |
| STD-003-CPP | High | Likely | Low | High | 5 |
| STD-004-CPP | Critical | Very Likely | Medium | Critical | 5 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | High | Unlikely | Low | Medium | 3 |
| STD-007-CPP | Medium | Likely | Low | Medium | 3 |
| STD-008-CPP | Medium | Likely | Low | Medium | 3 |
| STD-009-CPP | Medium | Possible | Low | Medium | 2 |
| STD-010-CPP | Medium | Possible | Low | Medium | 2 |
| N/A |  |  |  |  |  |

### 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 would be able to protect data stored on the physical devices, including databases, file systems, and backups. Green Pace would require that all the sensitive or personally identifiable data would be encrypted using AES-256 or higher whenever it is saved or archived. This makes sure that even if physical storage is invaded, the data would remain unreadable without the proper keys to decrypt. This policy applies to all the endpoints, databases, servers, and mobile devices storing company data. |
| Encryption in flight | Encryption in flight makes sure that data that is being transmitted across different networks is protected from interception or people messing with it. Green Pace requires the use of secure protocols including TLS 1.3, SSH, and VPNs from all internal and external data transmissions. This protects the important information from man-in-the-middle attacks or things such as eavesdropping. This policy applies to email communications, API interactions, remote work VPNs, inter-server communications. |
| Encryption in use | Encryption in use protects data while it is being processed in the memory. Methods such as secure enclaves and memory encryption make sure that even when the data is being controlled, sensitive information would stay confidential. Green Pace requires that any of the systems processing sensitive or regulated information would implement memory protection designs, and that the developers avoid showing data in unsecured temporary variables. This helps protect the data during runtime from memory scraping and unauthorized protection. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication makes sure that users and systems are properly verified before they can gain access. Green Pace makes sure there is MFA or multi-factor authentication for all of the internal and external access points, including user logins, administrative actions, and remote access. Strong password policies are also a requirement along with biometric authentication where it is possible, and certificate-based authentication for system-to-system communications that need to be implemented. |
| Authorization | Authorization shows and makes sure that authenticated users are allowed to access and modify things. Green Pace uses a role-based access control system or RBAC to ensure users only have the minimum required permissions necessary to do what they need to accomplish. Anything that needs to be added such as new users would need to be reviewed by the access control team and user levels would need to be looked over periodically, there would also need to be database changes that would only be allowed by authorized personnel. |
| Accounting | Accounting involves tracking and recording of the users’ actions to maintain system integrity and accountability. Green Pace requires that all access to important files, user logins, permission changes, and high priority operations be logged centrally. Logs need to be unchangeable, looked over constantly, and kept for review to meet the compliance standards. This ensures traceability of the users’ activities and helps with possible investigations. |

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

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

### Map the Principles

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

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

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

The Principles Table:

|  |  |  |
| --- | --- | --- |
| STD-001-CPP | 1,4 | Validates the input data to make sure the variables store the correct data safely. Keep It Simple help with choosing the right data types to avoid the code being too complex and helps to avoid unexpected problems. |
| STD-002-CPP | 1,5 | Validates the input data and protects it against out-of-bounds problems. Default Deny keeps operations from happening when values are out of the unexpected ranges, keeping it safe against invalid input. |
| STD-003-CPP | 7,9 | Sanitizes the data when sent to other systems to make sure string operations are safe and do not cause buffer overflows. Using Effective Quality Assurance Techniques can catch possible problems early in the testing process and help with validation. |
| STD-004-CPP | 7,3 | Sanitizes the data sent to other systems and protects databases by looking over the input data. Architect and design for the policies would make sure there is secure query handling that is included from the beginning. |
| STD-005-CPP | 8,9 | Practice of Defense in Depth adds a layer of protection even when there are memory issues, Use Effective Quality Assurance Techniques makes sure memory management vulnerabilities are found and stopped. |
| STD-006-CPP | 4,9 | Keep It Simple focuses on predictable error handling without depending on assertions, Use Effective Quality Assurance Techniques ensure runtime checks stay active while in production. |
| STD-007-CPP | 4,8 | Keep It Simple promotes easy to understand exception handling, Practice Defense In Depth allows it to catch exceptions safely and helps prevent system instability. |
| STD-008-CPP | 1,4 | Validate Input Data through things such as initializing carriable to avoid undefined behavior, Keep It Simple maintains the program’s reliability with known and initialized variables. |
| STD-009-CPP | 4,9 | Keep It Simple by avoiding goto statements to improve the codes maintainability, Use Effective Quality Assurance Techniques improves debugging and the logical consistency of the code. |
| STD-010-CPP | 8,9 | Practice Defense in Depth makes sure the files are properly closed to avoid any leaks, Use Effective Quality Assurance Techniques supports the auditing and keeps it stable using effective resource management. |

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 | 03/23/2025 | Module 3 Assignment | Cortland Burns |  |
| 1.2 | 4/13/2025 | Module 6 Project | Cortland Burns |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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