**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 | Ensures the input is normal and any abnormal inputs are flagged, and malicious data is prevented. |
| 1. Heed Compiler Warnings | Compiler warnings happen during the coding process. It allows developers to stop potential errors in cold and addressing it prevents potential security risks. |
| 1. Architect and Design for Security Policies | Software architecture and design needs to be considered when implementing security policies. |
| 1. Keep It Simple | Do not introduce unnecessary code that looks like an essay when a paragraph that does all the work is fine. This minimizes complexity and allows for better security. |
| 1. Default Deny | Automatically deny access unless permission is specified. |
| 1. Adhere to the Principle of Least Privilege | Individuals should not have more access to a system than what is needed to complete their task. This prevents security leaks. |
| 1. Sanitize Data Sent to Other Systems | Cleaning out useless and potentially harmful data from the system before passing more data in the system increases security. |
| 1. Practice Defense in Depth | Multiple layers of defense prevent attackers from easily accessing sensitive information. Keep in mind that too many layers would affect efficiency and that 3-5 layers are usually good enough. |
| 1. Use Effective Quality Assurance Techniques | Third-party is preferable due to their expertise and experience. They perform penetration testing and audits that ensures Quality Assurance. |
| 1. Adopt a Secure Coding Standard | Coding standards are rules that allow code security from the start. |

### 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 correct data type ensures that data is stored and processed accurately. This prevents issues such as overflow, underflow, or unintended type conversions. |

| **Noncompliant Code** |
| --- |
| The code uses an int data type to store a value that exceeds its maximum limit, leading to undefined behavior. |
| int largeValue = 2147483648; |

| **Compliant Code** |
| --- |
| The code uses a long long data type to store the large value, ensuring it fits within the data type's range. |
| long long largeValue = 2147483648; |

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

| **Principle 1) Validate Input Data:**  validating input ensures that data types are appropriate and prevent overflow or underflow.  **Principle 2) Heed Compiler Warnings:**  Compiler warnings about incorrect data types should be addressed to prevent vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | bugprone-too-small-loop-variable | Detects incorrect data type usage in loops |
| SonarQube | 10.8 | S3442 | Detects potential integer overflows |
| Coverity | 2024.12.0 | INTEGER\_OVERFLOW | Identifies integer overflow vulnerabilities |
| Cppcheck | 2.8 | uninitvar | Checks for uninitialized variables |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating data values ensures that inputs are within expected ranges or formats. This prevents vulnerabilities like buffer overflows, injection attacks, or unexpected behavior caused by invalid data. |

| **Noncompliant Code** |
| --- |
| The code accepts a numeric input without validation, which could lead to incorrect behavior if the input is outside the expected range. |
| #include <iostream>  using namespace std;  int main() {  int age;  cout << "Enter your age: ";  cin >> age; // No input validation  cout << "You are " << age << " years old." << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The code validates the input to ensure it is a positive integer within a reasonable range, preventing invalid or malicious input. |
| #include <iostream>  using namespace std;  int main() {  int age;  cout << "Enter your age: ";  while (!(cin >> age) || age < 0 || age > 120) { // Validate input  cin.clear(); // Clear error flags  cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Discard invalid input  cout << "Invalid input. Please enter a valid age (0-120): ";  }  cout << "You are " << age << " years old." << endl;  return 0;  } |

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

| **Principle 1) Validate Input Data:**  Ensure that all input data is checked for correctness, length, and type before processing  **Principle 7) Sanitize Data Sent to Other Systems:**  sanitizing data values prevents buffer overflows or injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | cert-err34-c | Checks for unsafe input handling |
| SonarQube | 10.8 | S108 | Identifies insecure input handling |
| Coverity | 2024.12.0 | BUFFER\_SIZE | Detects potential buffer overflows |
| Cppcheck | 2.8 | invalidscanf | Detects invalid input handling |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensuring proper handling of strings prevents vulnerabilities like buffer overflows, null-termination errors, and injection attacks. |

| **Noncompliant Code** |
| --- |
| The code accepts a numeric input without validation, which could lead to incorrect behavior if the input is outside the expected range. |
| #include <iostream>  using namespace std;  int main() {  int age;  cout << "Enter your age: ";  cin >> age; // No input validation  cout << "You are " << age << " years old." << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The code validates the input to ensure it is a positive integer within a reasonable range, preventing invalid or malicious input. |
| #include <iostream>  using namespace std;  int main() {  int age;  cout << "Enter your age: ";  while (!(cin >> age) || age < 0 || age > 120) { // Validate input  cin.clear(); // Clear error flags  cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Discard invalid input  cout << "Invalid input. Please enter a valid age (0-120): ";  }  cout << "You are " << age << " years old." << endl;  return 0;  } |

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

| **Principle 1) Validate input Data**  Ensures proper string handling and preventing vulnerabilities like buffer overflows, null-termination errors, and injection attacks.  **Principle 8) Defense in Depth**  This principle emphasizes multiple layers of security controls to protect against various attacks.  **Principle 10) Adopt a Secure Coding Standard**  Secure coding standards provide clear guidelines for writing secure code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | cert-err33-c | Checks for unsafe string handling |
| SonarQube | 10.8 | S108 | Identifies insecure string handling |
| Coverity | 2024.12.0 | BUFFER\_SIZE | Detects potential buffer overflows |
| Cppcheck | 2.8 | bufferAccessOutOfBounds | Detects potential buffer overflows |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection is a critical vulnerability that occurs when untrusted input is included in SQL queries without proper validation or sanitization. This allows attackers to manipulate databases, steal data, or execute unauthorized commands. |

| **Noncompliant Code** |
| --- |
| The code constructs an SQL query by directly concatenating user input, making it vulnerable to SQL injection attacks. |
| #include <iostream>  #include <mysql\_driver.h>  #include <mysql\_connection.h>  void queryDatabase(const std::string& userInput) {  sql::mysql::MySQL\_Driver \*driver;  sql::Connection \*con;  sql::Statement \*stmt;  driver = sql::mysql::get\_mysql\_driver\_instance();  con = driver->connect("tcp://127.0.0.1:3306", "user", "password");  con->setSchema("testdb");  stmt = con->createStatement();  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';"; // Vulnerable to SQL injection  stmt->execute(query);  delete stmt;  delete con;  } |

| **Compliant Code** |
| --- |
| The code uses prepared statements with parameterized queries to safely handle user input, preventing SQL injection. |
| #include <iostream>  #include <mysql\_driver.h>  #include <mysql\_connection.h>  void queryDatabase(const std::string& userInput) {  sql::mysql::MySQL\_Driver \*driver;  sql::Connection \*con;  sql::PreparedStatement \*pstmt;  driver = sql::mysql::get\_mysql\_driver\_instance();  con = driver->connect("tcp://127.0.0.1:3306", "user", "password");  con->setSchema("testdb");  pstmt = con->prepareStatement("SELECT \* FROM users WHERE username = ?"); // Parameterized query  pstmt->setString(1, userInput); // Safely bind user input  pstmt->execute();  delete pstmt;  delete con;  } |

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

| **Principle 8) Defense in Depth:**  Using parameterized queries and input validation prevents SQL injection.  **Principle 10) Adopting a Coding Standard:**  Adopting secure coding standards prevents SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.8 | S3649 | Detects SQL injection vulnerabilities |
| Coverity | 2024.12.0 | SQL\_INJECTION | Identifies insecure SQL query construction |
| Checkmarx | 9.4 | SQLi | Scans for SQL injection vulnerabilities |
| CodeQL | 2.10.5 | SQL injection | Detects SQL injection vulnerabilities |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Proper memory management prevents vulnerabilities such as memory leaks, buffer overflows, and use-after-free errors. |

| **Noncompliant Code** |
| --- |
| The code dynamically allocates memory but fails to deallocate it, resulting in a memory leak. |
| void processData() {  int\* data = new int[100]; // Memory allocated  // Process data  // Memory not deallocated  } |

| **Compliant Code** |
| --- |
| The code uses a smart pointer (std::unique\_ptr) to automatically manage memory allocation and deallocation, ensuring no memory leaks occur. |
| #include <memory>  void processData() {  auto data = std::make\_unique<int[]>(100); // Memory allocated and managed  // Process data  // Memory automatically deallocated when 'data' goes out of scope  } |

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

| **Principle 2) Heed Compiler Warnings:**  Helps to catch memory-related issues early in the development process.  **Principle 7) Sanitize Data Sent to Other Systems:**  Ensures that memory operations are safe and free from malicious input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | cppcoreguidelines-owning-memory | Ensures proper memory ownership |
| SonarQube | 10.8 | S3649 | Detect memory leaks and improper memory management |
| Coverity | 2024.12.0 | RESOURCE\_LEAK | Identifies memory leaks and resource management issues |
| Valgrind | 3.24.0 | memorycheck | Detects memory leaks and invalid memory usage |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are used to verify assumptions during development and debugging. They help catch logical errors early but should not be used for runtime error handling, as they are typically disabled in production builds. Proper use of assertions ensures robust and maintainable code. |

| **Noncompliant Code** |
| --- |
| The code uses an assertion to validate user input, which is inappropriate because assertions are not active in production builds. |
| #include <cassert>  #include <iostream>  void processInput(int value) {  assert(value > 0); // Incorrect use of assertion for input validation  std::cout << "Processing value: " << value << std::endl;  } |

| **Compliant Code** |
| --- |
| The code uses proper input validation instead of assertions to handle user input, ensuring the program behaves correctly in all build configurations. |
| #include <iostream>  #include <stdexcept>  void processInput(int value) {  if (value <= 0) {  throw std::invalid\_argument("Value must be greater than 0."); // Proper input validation  }  std::cout << "Processing value: " << value << std::endl;  } |

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

| **Principle 2) Head Compiler Warning:**  Heeding warnings about improper use of assertions ensures they are only used for debugging.  **Principle 4) Keep it Simple:**  Using assertions only for debugging avoids unnecessary complexity in runtime error handling.  **Principle 9) Effective Quality Assurance:**  Testing assertions ensure they are used appropriately. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | misc-assert | Detects misuse of assertions |
| SonarQube | 10.8 | S5905 | Flags assertions used for runtime checks |
| Cppcheck | 2.8 | assertionFail | Identifies improper use of assertions |
| Coverity | 2024.12.0 | ASSERT\_SIDE\_EFFECT | Detects side effects in assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Proper exception handling ensures that errors are managed gracefully, preventing crashes, data corruption, or security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The code does not handle exceptions, which could lead to resource leaks or undefined behavior if an error occurs. |
| #include <iostream>  void riskyOperation() {  int\* resource = new int[100]; // Resource allocated  throw std::runtime\_error("An error occurred!"); // Exception thrown  delete[] resource; // Never executed, causing a memory leak  }  int main() {  riskyOperation();  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses RAII (via std::unique\_ptr) to ensure resources are cleaned up even if an exception is thrown. |
| #include <iostream>  #include <memory>  void riskyOperation() {  auto resource = std::make\_unique<int[]>(100); // Resource managed by smart pointer  throw std::runtime\_error("An error occurred!"); // Exception thrown  // Resource automatically cleaned up when 'resource' goes out of scope  }  int main() {  try {  riskyOperation();  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principle 2) Heed Compiler Warnings:**  Helps to catch memory-related issues early in the development process.  **Principle 7) Sanitize Data Sent to Other Systems:**  Ensures that memory operations are safe and free from malicious input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | cppcoreguidelines-owning-memory | Ensures proper memory ownership. |
| SonarQube | 10.8 | S3649 | Detect memory leaks and improper memory management. |
| Coverity | 2024.12.0 | RESOURCE\_LEAK | Identifies memory leaks and resource management issues. |
| Valgrind | 3.24.0 | memorycheck | Detects memory leaks and invalid memory usage. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | [STD-008-CPP] | Input validation ensures that all user-provided data is checked for correctness, length, and type before processing. |

| **Noncompliant Code** |
| --- |
| The code accepts user input without validation, which could lead to buffer overflows or unexpected behavior if the input exceeds the expected size. |
| #include <iostream>  using namespace std;  int main() {  char name[10];  cout << "Enter your name: ";  cin >> name; // No input validation  cout << "Hello, " << name << "!" << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The code validates the length of user input to ensure it does not exceed the buffer size, preventing buffer overflows. |
| #include <iostream>  #include <string>  using namespace std;  int main() {  string name;  cout << "Enter your name: ";  getline(cin, name); // Use getline for safer input handling  if (name.length() > 9) {  cout << "Name is too long. Please enter a name with 9 or fewer characters." << endl;  } else {  cout << "Hello, " << name << "!" << endl;  }  return 0;  } |

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

| **Principle 1) Input Validation:**  Ensures that all input data is checked for correctness, length, and type before processing.  **Principle 7) Sanitize Data Sent to Other Systems:**  Sanitizing input ensures that data passed to other systems is safe. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | cert-err34-c | Checks for unsafe input handling |
| SonarQube | 10.8 | S108 | Identifies insecure input handling |
| Coverity | 2024.12.0 | BUFFER\_SIZE | Detects potential buffer overflows |
| Cppcheck | 2.8 | invalidscanf | Detects invalid input handling |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure File Handling | [STD-009-CPP] | Secure file handling ensures that files are accessed, read, and written to in a way that prevents unauthorized access, data corruption, or injection attacks. |

| **Noncompliant Code** |
| --- |
| The code opens a file without validating the file path or checking for errors, which could lead to security vulnerabilities such as path traversal attacks or crashes. |
| #include <iostream>  #include <fstream>  using namespace std;  void readFile(const string& filename) {  ifstream file(filename); // No validation of file path  string line;  while (getline(file, line)) {  cout << line << endl;  }  } |

| **Compliant Code** |
| --- |
| The code validates the file path and checks for errors before attempting to read the file, ensuring secure and robust file handling. |
| #include <iostream>  #include <fstream>  #include <stdexcept>  using namespace std;  void readFile(const string& filename) {  // Validate file path (e.g., ensure it does not contain "../")  if (filename.find("..") != string::npos) {  throw invalid\_argument("Invalid file path."); // Prevent path traversal  }  ifstream file(filename);  if (!file.is\_open()) {  throw runtime\_error("Failed to open file."); // Handle file open errors  }  string line;  while (getline(file, line)) {  cout << line << endl;  }  } |

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

| **Principle 4) Keep it Simple:**  Simplifying file handling logic reduces the risk of vulnerabilities.  **Principle 5) Default Deny:**  File access is denied by default secures file handling..  **Principle 9) Default Deny:**  Testing file handling ensures that vulnerabilities like path traversal are prevented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | cert-err33-c | Checks for unsafe file handling |
| SonarQube | 10.8 | S5042 | Identifies insecure file path handling |
| Coverity | 2024.12.0 | PATH\_MANIPULATION | Detects potential file path manipulation |
| Cppcheck | 2.8 | fileOpen | Detects potential file handling issues |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Proper Error Handling | [STD-010-CPP] | Proper error handling ensures that unexpected conditions are managed gracefully, preventing crashes, data corruption, or security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The code does not handle errors, which could lead to crashes or undefined behavior if an exception occurs. |
| #include <iostream>  int divide(int a, int b) {  return a / b; // No error handling for division by zero  }  int main() {  std::cout << divide(10, 0) << std::endl; // Potential crash  return 0;  } |

| **Compliant Code** |
| --- |
| The code checks for division by zero and handles the error gracefully, ensuring the program does not crash. |
| #include <iostream>  #include <stdexcept>  int divide(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Division by zero is not allowed."); // Handle error  }  return a / b;  }  int main() {  try {  std::cout << divide(10, 0) << std::endl;  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl; // Graceful error handling  }  return 0;  } |

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

| **Principle 8) Defense in Depth:**  Adding multiple layers of error handling ensures system stability.  **Principle 10) Adopt a Secure Coding Standard**  Following secure coding standards ensures robust error handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | misc-throw-by-value-catch-by-reference | Ensures proper error handling |
| SonarQube | 10.8 | S3981 | Flags unhandled exceptions |
| Coverity | 2024.12.0 | UNCAUGHT\_EXCEPTION | Detects uncaught exceptions |
| Cppcheck | 2.8 | uninitvar | Detects uninitialized variables in error 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.

There are two main phases of Automation, Pre-production and Production. In the Pre-production phase, Assess and Plan is the first step, where teams evaluate possible threats and regulations while responding to threats. The next step is the Design stage, where using best practices like OWASP builds a strong foundation for development. The next stage is Build, where software development uses secure and trusted repositories along with trusted open-source libraries. The next step is the Verify and test Stage, where vulnerability scanning, compliance checks, and functional security testing are performed for software integrity.

In the Production phase, the first step is Transition and Health Check, which configures deployment environments, applies security settings, and conducts penetration testing. The next stage is Monitor and Detect, where real-time security monitoring, logging, and intrusion detection are performed. If threats are detected, the next stage, Respond will be initiated to block attacks and resolve the issue. The Final stage is Maintain and Stabilize, where it ensures that systems are continually assessed against security baselines and restored to a stable state following any security incidents.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Medium | Low | High | 2 |
| STD-002-CPP | High | Medium | Low | High | 3 |
| STD-003-CPP | High | High | Medium | High | 4 |
| STD-004-CPP | High | High | Medium | High | 4 |
| STD-005-CPP | High | High | High | High | 5 |
| STD-006-CPP | Medium | Medium | Medium | Medium | 3 |
| STD-007-CPP | High | High | High | High | 5 |
| STD-008-CPP | High | Medium | Low | High | 3 |
| STD-009-CPP | High | Medium | Medium | High | 4 |
| STD-010-CPP | High | High | Low | High | 3 |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 | * Protect data by encrypting it. * Applied to all sensitive data stored on servers, cloud storage, and endpoints. Must be enabled for databases, file systems, and backups. * Prevents unauthorized access if physical or digital storage is compromised. |
| Encryption in flight | * Secures data transmission to prevent interception. * Required for all web traffic, API calls, database connections, and internal communications. * Protects against eavesdropping, and data leaks during transmission. |
| Encryption in use | * Ensures data remains encrypted while being processed in memory. * Applied when handling sensitive data in applications. * Prevents memory-scraping attacks and unauthorized access during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | * Verifies user identity before granting access. * Required for all user logins, API access, and remote connections. Use of MFA and constant password resets. * Prevents unauthorized access and credential-based attacks. |
| Authorization | * Determines what authenticated users can access. * Applied when assigning permissions. * Limits exposure to sensitive data and reduces insider threat risks. |
| Accounting | * Logs and audits user activities. * Tracks all critical actions: user logins, database changes, file access, and privilege escalations. Logs must be immutable and retained. * Enables investigations and detection of suspicious behavior. |

### 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

| Rule | Principles | Explanation |
| --- | --- | --- |
| STD-001-CPP | 1  2 | **Principle 1) Validate Input Data:**  validating input ensures that data types are appropriate and prevent overflow or underflow.  **Principle 2) Heed Compiler Warnings:**  Compiler warnings about incorrect data types should be addressed to prevent vulnerabilities. |
| STD-002-CPP | 1  7 | **Principle 1) Validate Input Data:**  Ensure that all input data is checked for correctness, length, and type before processing  **Principle 7) Sanitize Data Sent to Other Systems:**  sanitizing data values prevents buffer overflows or injection attacks. |
| STD-003-CPP | 1  8  10 | **Principle 1) Validate input Data**  Ensures proper string handling and preventing vulnerabilities like buffer overflows, null-termination errors, and injection attacks.  **Principle 8) Defense in Depth**  This principle emphasizes multiple layers of security controls to protect against various attacks.  **Principle 10) Adopt a Secure Coding Standard**  Secure coding standards provide clear guidelines for writing secure code. |
| STD-004-CPP | 8  10 | **Principle 8) Defense in Depth:**  Using parameterized queries and input validation prevents SQL injection.  **Principle 10) Adopting a Coding Standard:**  Adopting secure coding standards prevents SQL injection. |
| STD-005-CPP | 2  7 | **Principle 2) Heed Compiler Warnings:**  Helps to catch memory-related issues early in the development process.  **Principle 7) Sanitize Data Sent to Other Systems:**  Ensures that memory operations are safe and free from malicious input. |
| STD-006-CPP | 2  4  9 | **Principle 2) Head Compiler Warning:**  Heeding warnings about improper use of assertions ensures they are only used for debugging.  **Principle 4) Keep it Simple:**  Using assertions only for debugging avoids unnecessary complexity in runtime error handling.  **Principle 9) Effective Quality Assurance:**  Testing assertions ensure they are used appropriately. |
| STD-007-CPP | 2  7 | **Principle 2) Heed Compiler Warnings:**  Helps to catch memory-related issues early in the development process.  **Principle 7) Sanitize Data Sent to Other Systems:**  Ensures that memory operations are safe and free from malicious input. |
| STD-008-CPP | 1  7 | **Principle 1) Input Validation:**  Ensure that all input data is checked for correctness, length, and type before processing.  **Principle 7) Sanitize Data Sent to Other Systems:**  Sanitizing input ensures that data passed to other systems is safe. |
| STD-009-CPP | 4  5  9 | **Principle 4) Keep it Simple:**  Simplifying file handling logic reduces the risk of vulnerabilities.  **Principle 5) Default Deny:**  File access is denied by default secures file handling.  **Principle 9) Default Deny:**  Testing file handling ensures that vulnerabilities like path traversal are prevented. |
| STD-010-CPP | 8  10 | **Principle 8) Defense in Depth:**  Adding multiple layers of error handling ensures system stability.  **Principle 10) Adopt a Secure Coding Standard**  Following secure coding standards ensures robust error handling. |

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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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