**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 validate any data input by users or systems to ensure it is safe and correct, thereby preventing attacks such as SQL injection. |
| 1. Heed Compiler Warnings | Consider the warnings issued by the compiler during the development phase. These warnings often indicate potential coding errors or vulnerabilities that could be exploited. |
| 1. Architect and Design for Security Policies | Incorporate security principles from the initial phase, integrating them into the system architecture rather than adding them later as an afterthought. |
| 1. Keep It Simple | Simplicity in design and implementation reduces the risk of introducing new security flaws, as complex systems are harder to understand and secure. |
| 1. Default Deny | Set default permissions to deny access, only allowing it when explicitly granted. This principle limits exposure by ensuring that only authorized users have access to resources. |
| 1. Adhere to the Principle of Least Privilege | Grant users and systems the minimal level of access necessary to perform their functions. This limits the potential for human error and reduces the risk of damage from security breaches. |
| 1. Sanitize Data Sent to Other Systems | Clean and validate data before sending it to other systems to prevent the spread of malicious inputs. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls and defenses throughout the system. This approach ensures that if one layer fails, other layers will still protect the system. |
| 1. Use Effective Quality Assurance Techniques | Ensure a strong authentication mechanism to verify the identities of users and processes and implement strict authorization policies to control access to resources. |
| 1. Adopt a Secure Coding Standard | Continuously log and monitor system activities to detect and respond to potential security incidents. Effective logging and monitoring help identify unusual behavior and facilitate a more robust incident response. |

### 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] | Ensuring the proper use of data types |

| **Noncompliant Code** |
| --- |
| Using inappropriate data types for variables, which can lead to inefficient memory usage or errors. |
| // Using an int for a variable that will only hold small positive values  int age = 25;  // Using a double for a variable that should be an integer  double itemCount = 10;  // Using an unsigned int where signed int is expected  unsigned int count = -5; |

| **Compliant Code** |
| --- |
| Using appropriate data types that fit the expected range and usage of the variables, ensuring efficient memory usage and preventing type-related errors. |
| // Using unsigned char for a small positive value  unsigned char age = 25;  // Using int for an integer value  int itemCount = 10;  // Using int for signed values  int count = -5; |

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

| **Principles(s):** Ensure the efficient memory usage, This principle emphasizes using appropriate data types to match the expected range and usage of variables. It prevents type-related errors and optimizes memory usage. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Cppcheck | Cppcheck is a static tool for C/C++ code. It detects the types of bugs that the compilers normally do not detect. |
| PSV-Studio | 7.13 | PVS-Studio Analyzer | PVS-Studio is a static code analyzer for detecting bugs and potential vulnerabilities in the source code of programs. |
| Helix QAC | 2021.2 | C++3013 | Helix QAC is a static code analysis tool for C and C++ languages that helps ensure compliance with coding standards and improves code quality. |
| PRQA AQ-C++ | 4.4 | 3013 | PRQA QA-C++ is a static analysis tool that enforces coding standards and best practices, identifying issues in C++ code to improve code quality and maintainability. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Initialize and Validate Data Values |

| **Noncompliant Code** |
| --- |
| Failing to initialize variables or validate their values before using them, which can lead to unpredictable behavior or security issues. |
| // Uninitialized variable usage  int totalItems;  totalItems += 10;  // No validation of user input  int userAge;  std::cin >> userAge;  std::cout << "User age is: " << userAge << std::endl; |

| **Compliant Code** |
| --- |
| Properly initializing variables and validating data values to ensure they are within expected ranges before using them. |
| // Initializing variable  int totalItems = 0;  totalItems += 10;  // Validation of user input  int userAge = 0;  std::cout << "Enter your age: ";  std::cin >> userAge;  if (userAge < 0 || userAge > 120) {  std::cerr << "Invalid age entered!" << std::endl;  } else {  std::cout << "User age is: " << userAge << std::endl;  } |

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

| **Principles(s):** Ensuring data integrity and preventing errors. This principle focuses on the necessity of initializing variables and validating data values to prevent unpredictable behavior and potential security issues. Proper initialization and validation help maintain data integrity and ensure the program operates as expected. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Assert\_failure | Astree is a static analysis tool designed to prove the absence of runtime errors in embedded control software. |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | LDRA tool suite is a set of software analysis and testing tools for verifying the quality and security of code in safety-critical applications. |
| Klocwork | 2021.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT | Klocwork is a static code analysis tool used to identify security, safety, and coding standards issues. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensuring String Correctness |

| **Noncompliant Code** |
| --- |
| Using C-style strings without proper bounds checking or safe string handling functions, leading to potential buffer overflows and undefined behavior. |
| #include <cstring>  #include <iostream>  // Using a fixed-size buffer without bounds checking  void unsafeStringCopy(const char\* src) {  char buffer[10];  std::strcpy(buffer, src); // Potential buffer overflow  std::cout << "Buffer contains: " << buffer << std::endl;  }  int main() {  const char\* longString = "This is a very long string";  unsafeStringCopy(longString);  return 0;  } |

| **Compliant Code** |
| --- |
| Using C++ standard library string functions and classes, such as ‘std::string’, to handle strings safely and efficiently, preventing buffer overflows and other string-related issues. |
| #include <string>  #include <iostream>  // Using std::string for safe string operations  void safeStringCopy(const std::string& src) {  std::string buffer = src.substr(0, 9); // Ensuring buffer size limit  std::cout << "Buffer contains: " << buffer << std::endl;  }  int main() {  std::string longString = "This is a very long string";  safeStringCopy(longString);  return 0;  } |

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

| **Principles(s):** Ensuring data safety and preventing undefined behavior. This principle focuses on using safe string handling functions and proper bounds checking to avoid buffer overflows and undefined behavior. By ensuring string correctness, the code becomes more secure and stable. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Assert\_failure | Astree is a static analysis tool designed to prove the absence of runtime errors in embedded control software. |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | LDRA tool suite is a set of software analysis and testing tools for verifying the quality and security of code in safety-critical applications. |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-STR51-a | Parasoft C/C++test is a static analysis and unit testing tool for C and C++ applications, helping ensure code quality and adherence to coding standards. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++- INT50 | Axivion Bauhaus Suite is a comprehensive toolset for static code analysis, detecting code smells, and enforcing coding standards. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| Constructing SQL queries by directly concatenating user input, which makes the application vulnerable to SQL injection attacks. |
| // Vulnerable to SQL injection  std::string userInput = "example'; DROP TABLE users; --";  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  mysql\_query(conn, query.c\_str()); |

| **Compliant Code** |
| --- |
| Using prepared statements to safely handle user input and prevent SQL injection attacks. |
| // Using prepared statements to prevent SQL injection  const char\* query = "SELECT \* FROM users WHERE username = ?";  MYSQL\_STMT\* stmt = mysql\_stmt\_init(conn);  mysql\_stmt\_prepare(stmt, query, strlen(query));  MYSQL\_BIND bind[1];  memset(bind, 0, sizeof(bind));  bind[0].buffer\_type = MYSQL\_TYPE\_STRING;  bind[0].buffer = (char\*)userInput.c\_str();  bind[0].buffer\_length = userInput.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):** Ensuring secure handling of user input to prevent SQL injection. This principle emphasizes using prepared statements and parameterized queries to safely handle user input, thereby preventing SQL injection attacks. By doing so, it ensures that user input does not alter the structure of SQL queries, maintaining database integrity and security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Assert\_failure | Astree is a static analysis tool designed to prove the absence of runtime errors in embedded control software. |
| SQLMap | 1.5.6 | SQL Injection Checker | SQLMap is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over database servers. |
| SonarQube | 8.9 | Security Hotspots | SonarQube is a tool for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Ensure Proper Memory Management |

| **Noncompliant Code** |
| --- |
| Manually managing dynamic memory without proper checks, leading to potential memory leaks and undefined behavior. |
| void unsafeFunction() {  int\* array = new int[10]; // Dynamic allocation  // Using array without boundary checks  for (int i = 0; i < 20; ++i) { // Buffer overflow risk  array[i] = i;  }  // Forgetting to deallocate memory  } |

| **Compliant Code** |
| --- |
| Using smart pointers and standard library containers to manage dynamic memory safely, ensuring automatic deallocation and preventing buffer overflows. |
| void safeFunction() {  auto array = std::make\_unique<std::vector<int>>(10); // Using smart pointer and vector  // Using array with boundary checks  for (size\_t i = 0; i < array->size(); ++i) {  (\*array)[i] = i;  }  // Memory is automatically managed and deallocated  } |

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

| **Principles(s):** Ensuring proper memory management. This principle focuses on using modern C++ features like smart pointers and standard library containers to manage dynamic memory safely. It ensures that memory is automatically deallocated, preventing memory leaks and buffer overflows, which leads to more reliable and maintainable code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MEM51-a | Parasoft C/C++test is a static analysis and unit testing tool for C and C++ applications, helping ensure code quality and adherence to coding standards. |
| Coverity | 2021.3 | MEM\_LEAK | Coverity is a static analysis tool that identifies software bugs and security vulnerabilities. |
| Valgrind | 3.17.0 | Memcheck | |  | | --- | |  |   Valgrind is an instrumentation framework for building dynamic analysis tools, used primarily for detecting memory management and threading bugs. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use Assertions to Enforce Assumptions |

| **Noncompliant Code** |
| --- |
| Failing to use assertions to validate critical assumptions, leading to potential undetected logical errors. |
| #include <iostream>  int divide(int a, int b) {  // No check to ensure b is not zero  return a / b;  }  int main() {  int result = divide(10, 0); // Division by zero  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using assertions to validate assumptions, such as ensuring that the divisor is not zero before performing division. |
| #include <iostream>  #include <cassert>  int divide(int a, int b) {  assert(b != 0 && "Division by zero!"); // Assertion to ensure b is not zero  return a / b;  }  int main() {  int result = divide(10, 2); // Valid division  std::cout << "Result: " << result << std::endl;  } |

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

| **Principles(s):** Using assertions to enforce assumptions. This principle focuses on the importance of using assertions to validate critical assumptions in the code. Assertions help detect logical errors early by ensuring that certain conditions hold true before the code proceeds. This practice leads to more robust and reliable code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1 | Assertion Checker | CodeSonar is a static analysis tool for source code and binaries, designed for safety-critical applications and helps find security vulnerabilities and reliability issues. |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-ERR33-c | Parasoft C/C++test is a static analysis and unit testing tool for C and C++ applications, helping ensure code quality and adherence to coding standards. |
| Coverity | 2021.3 | ASSERT\_FAIL | Coverity is a static analysis tool that identifies software bugs and security vulnerabilities. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++- ERR30 | Axivion Bauhaus Suite is a comprehensive toolset for static code analysis, detecting code smells, and enforcing coding standards. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle Exceptions Properly |

| **Noncompliant Code** |
| --- |
| Throwing a generic exception or a string as an exception is not recommended. It provides little information about the type of error and can lead to difficulties in error handling. |
| // Noncompliant: Throwing a generic exception  void doSomething() {  if (errorCondition) {  throw "An error occurred"; // Noncompliant  }  } |

| **Compliant Code** |
| --- |
| Using a specific exception type like ‘std::runtime\_error’ gives more context about the error, making it easier to handle and debug. |
| // Compliant: Throwing a specific exception type  #include <stdexcept>  void doSomething() {  if (errorCondition) {  throw std::runtime\_error("An error occurred"); // Compliant  }  } |

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

| **Principles(s):** Handling exceptions properly. This principle emphasizes using specific exception types rather than generic ones to provide more context about errors. It ensures that errors are easier to handle and debug, leading to more robust error management and better maintainability of the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | High | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1 | Exception Handling | CodeSonar is a static analysis tool for source code and binaries, designed for safety-critical applications and helps find security vulnerabilities and reliability issues. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++- ERR50 | Axivion Bauhaus Suite is a comprehensive toolset for static code analysis, detecting code smells, and enforcing coding standards. |
| PC-lint Plus | 9.0 | 1712 | PC-lint Plus is a static code analysis tool that identifies bugs, glitches, inconsistencies, and violations of coding standards. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Sensitive Data | [STD-008-CPP] | Secure Handling of Sensitive Data |

| **Noncompliant Code** |
| --- |
| Storing sensitive data such as passwords in plain text is a security risk. If the data is compromised, attackers can easily read and misuse the information. In this example, the password is stored in plain text. |
| // Noncompliant: Storing sensitive data in plain text  #include <string>  void storePassword(const std::string& password) {  std::string storedPassword = password; // Noncompliant: Plain text storage  // Store the password (simulated)  }  int main() {  std::string userPassword = "MySecretPassword";  storePassword(userPassword);  return 0;  } |

| **Compliant Code** |
| --- |
| Storing sensitive data securely involves hashing passwords before storage. In this example, the password is hashed using SHA-256 before being stored, making it more secure. Even if the data is compromised, attackers cannot easily recover the original password. |
| // Compliant: Storing sensitive data securely using hashing  #include <string>  #include <iostream>  #include <openssl/sha.h> // Example using OpenSSL for hashing  std::string hashPassword(const std::string& password) {  unsigned char hash[SHA256\_DIGEST\_LENGTH];  SHA256(reinterpret\_cast<const unsigned char\*>(password.c\_str()), password.size(), hash);    std::string hashedPassword;  for (unsigned char i : hash) {  char buf[3];  snprintf(buf, sizeof(buf), "%02x", i);  hashedPassword += buf;  }  return hashedPassword;  } |

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

| **Principles(s):** Secure handling of sensitive data. This principle focuses on the importance of securely handling sensitive data, such as passwords. Storing passwords in plain text is a major security risk. Instead, passwords should be hashed using strong cryptographic algorithms before storage to ensure that even if the data is compromised, the passwords cannot be easily recovered. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Security Hotspots | SonarQube is a tool for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities. |
| CodeSonar | 7.1 | Sensitive Data | CodeSonar is a static analysis tool for source code and binaries, designed for safety-critical applications and helps find security vulnerabilities and reliability issues. |
| Fortify Static Code Analyzer | 20.2 | Data Flow Analyzer | Fortify Static Code Analyzer identifies security vulnerabilities in source code by using data flow analysis to trace paths through the code and identify potential issues. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | [STD-009-CPP] | Secure Error Handling |

| **Noncompliant Code** |
| --- |
| Exposing detailed error messages to end users can reveal sensitive information about the system, which can be exploited by attackers. In this example, the error message reveals the filename and the nature of the error. |
| // Noncompliant: Exposing detailed error messages  #include <iostream>  #include <fstream>  #include <string>  void readFile(const std::string& filename) {  std::ifstream file(filename);  if (!file.is\_open()) {  std::cerr << "Error: Unable to open file '" << filename << "'. Please check if the file exists and try again." << std::endl;  return;  }  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }  int main() {  std::string filename;  std::cout << "Enter filename: ";  std::cin >> filename;  readFile(filename);  return 0;  } |

| **Compliant Code** |
| --- |
| Providing generic error messages to end users helps prevent the disclosure of sensitive system information. In this example, detailed error information is logged internally, while the user receives a generic message. |
| // Compliant: Providing generic error messages to users  #include <iostream>  #include <fstream>  #include <string>  #include <stdexcept>  void readFile(const std::string& filename) {  std::ifstream file(filename);  if (!file.is\_open()) {  // Log detailed error internally  throw std::runtime\_error("Error: Unable to open file '" + filename + "'");  }  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }  int main() {  std::string filename;  std::cout << "Enter filename: ";  std::cin >> filename;  try {  readFile(filename);  } catch (const std::exception& e) {  std::cerr << "An error occurred while processing your request. Please try again later." << std::endl;  }  return 0;  } |

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

| **Principles(s):** Secure error handling. This principle emphasizes the importance of providing generic error messages to end users while logging detailed error information internally. This approach helps prevent the disclosure of sensitive system information that could be exploited by attackers, thereby enhancing the security of the application. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Security Hotspots | SonarQube is a tool for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++- ERR30 | Axivion Bauhaus Suite is a comprehensive toolset for static code analysis, detecting code smells, and enforcing coding standards. |
| Fortify Static Code Analyzer | 20.2 | Data Flow Analyzer | Fortify Static Code Analyzer identifies security vulnerabilities in source code by using data flow analysis to trace paths through the code and identify potential issues. |
| CodeSonar | 7.1 | Error Handling | CodeSonar is a static analysis tool for source code and binaries, designed for safety-critical applications and helps find security vulnerabilities and reliability issues. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Operations | [STD-010-CPP] | Secure File Operations |

| **Noncompliant Code** |
| --- |
| Opening files without proper validation and error handling can lead to various security issues, such as accessing or modifying unintended files. In this example, the code attempts to open a file without verifying its legitimacy or handling potential errors securely. |
| // Noncompliant: Opening files without proper checks  #include <iostream>  #include <fstream>  void readFile(const std::string& filename) {  std::ifstream file(filename);  if (!file) {  std::cerr << "Error: Unable to open file " << filename << std::endl;  return;  }  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }  int main() {  std::string filename = "data.txt";  readFile(filename); // Noncompliant: Potentially unsafe file operations  return 0;  } |

| **Compliant Code** |
| --- |
| Securely handling file operations involves validating the file path, using appropriate file modes, and handling errors properly. In this example, the file is opened in a secure mode with proper error checking, and exceptions are used to handle potential issues. |
| // Compliant: Securely handling file operations with proper checks  #include <iostream>  #include <fstream>  #include <stdexcept>  void readFile(const std::string& filename) {  std::ifstream file(filename, std::ios::in | std::ios::binary);  if (!file) {  throw std::runtime\_error("Error: Unable to open file " + filename);  }  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }  int main() {  std::string filename = "data.txt";  try {  readFile(filename); // Compliant: Secure file operations with proper error handling  } catch (const std::exception& e) {  std::cerr << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Secure file operations. This principle focuses on validating file paths, using appropriate file modes, and handling errors properly when performing file operations. Properly handling file operations prevents security issues, such as accessing or modifying unintended files, and ensures the application operates reliably. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P3 | P4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 20.2 | Data Flow Analyzer | Fortify Static Code Analyzer identifies security vulnerabilities in source code by using data flow analysis to trace paths through the code and identify potential issues. |
| SonarQube | 8.9 | Security Hotspots | SonarQube is a tool for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities. |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-FIO02-a | Parasoft C/C++test is a static analysis and unit testing tool for C and C++ applications, helping ensure code quality and adherence to coding standards. |

### 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.

To ensure enforcement and compliance with the standards defined in this policy, Green Pace will integrate security practices into the existing DevOps process, transitioning to a DevSecOps model. Automated testing continues with integrity checks and defense-in-depth measures to ensure ongoing prevention. Methods such as network monitoring, penetration testing, and performance logging are employed for continuous threat detection. Similar to QA testing, security testing should be performed early and frequently to maintain robust security.

### 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 | Medium | Medium | Medium | 1 |
| STD-003-CPP | High | Medium | Medium | High | 3 |
| STD-004-CPP | High | Likely | Medium | Medium | 1 |
| STD-005-CPP | High | Medium | Medium | Medium | 1 |
| STD-006-CPP | Low | Unlikely | Low | Low | 3 |
| STD-007-CPP | Low | High | Medium | Low | 2 |
| STD-008-CPP | High | Likely | Medium | Medium | 1 |
| STD-009-CPP | High | Medium | Low | High | 2 |
| STD-010-CPP | High | Medium | Medium | Medium | 4 |

### 

### 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 refers to the practice of encrypting data that is stored on physical or virtual storage devices. The policy mandates that sensitive data must be encrypted when stored to protect it from unauthorized access. This ensures that even if physical devices are compromised, the data remains secure and unreadable without the encryption key. |
| Encryption in flight | Encryption in flight or in transit refers to encrypting data while it is being transmitted across networks. The policy requires that sensitive data must be encrypted during transmission to prevent interception by unauthorized parties. This protects data integrity and confidentiality as it moves between systems, users, and applications. |
| Encryption in use | Encryption in use refers to the encryption of data while it is being processed or used in memory. The policy enforces encryption of sensitive data even when it is actively being processed to safeguard it against threats such as memory scraping or unauthorized access during execution. This ensures comprehensive protection of data at all stages. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user, device, or system. The policy requires robust authentication mechanisms to ensure that only authorized users can access sensitive systems and data. This prevents unauthorized access and helps maintain the integrity and confidentiality of the system. |
| Authorization | Authorization is the process of determining what an authenticated user is allowed to do within the system. The policy ensures that users have access only to the resources and data necessary for their roles. This principle of least privilege minimizes the risk of accidental or malicious access to sensitive information. |
| Accounting | Accounting involves tracking user activities and access to resources within the system. The policy mandates detailed logging and monitoring of user actions to detect and respond to suspicious activities. This accountability helps in forensic analysis and ensures compliance with regulatory requirements. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 06/16/2024 | Initial Template | Mohamed Jaddour | [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 |