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



# Green Pace Secure Development Policy

# Developer: Steven Rodas

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_4d34og8)

[C/C++ Ten Coding Standards 3](#_2s8eyo1)

[Coding Standard 1 4](#_3rdcrjn)

[Coding Standard 2 5](#_26in1rg)

[Coding Standard 3 6](#_lnxbz9)

[Coding Standard 4 7](#_35nkun2)

[Coding Standard 5 8](#_1ksv4uv)

[Coding Standard 6 9](#_2jxsxqh)

[Coding Standard 7 10](#_z337ya)

[Coding Standard 8 11](#_28h4qwu)

[Coding Standard 9 13](#_nmf14n)

[Coding Standard 10 14](#_37m2jsg)

[Defense-in-Depth Illustration 15](#_1mrcu09)

[Project One 15](#_46r0co2)

[1.](#_2lwamvv) Revise the C/C++ Standards 15

[2.](#_111kx3o) Risk Assessment 15

[3.](#_3l18frh) Automated Detection 15

[4.](#_206ipza) Automation 15

[5.](#_4k668n3) Summary of Risk Assessments 16

[6.](#_2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_3ygebqi)

[Enforcement 18](#_2dlolyb)

[Exceptions Process 18](#_sqyw64)

[Distribution 19](#_3cqmetx)

[Policy Change Control 19](#_1rvwp1q)

[Policy Version History 19](#_4bvk7pj)

[Appendix A Lookups 19](#_2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_1664s55)

## 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 | Validating input data is one of the most important security principles to implement. Validating input data ensures that any data that the user provides to the program, or from other external sources, must adhere to the expected format, type and other constraints set forth by the developers. This is of importance since improper input validation leads to huge security vulnerabilities that can lead to buffer overflows, SQL injections and other attacks. |
| 1. Heed Compiler Warnings | Compiler Warnings are warnings that are prompted to developers through the console upon running or testing code. These warnings are important indicators and can tell you if there are potential issues or areas of concern in the code. These warnings can include possible bugs and insecure coding practices. It is wise to address these warnings as they are valuable warnings every developer should consider. |
| 1. Architect and Design for Security Policies | Thinking through the design and implementation of security policies is a crucial step to securing your software. Through early implementation of security policies earlier in the design phase, you can potentially filter out potential vulnerabilities that are usually hidden earlier through development phases that could show up as a larger problem later on in the development process. |
| 1. Keep It Simple | Simplicity is a core security principle that reduces the risks of potential vulnerabilities. Simply put, complex and large systems are a lot harder to understand, test, and secure, which in turn makes them more prone to bugs and other security flaws. To combat this, keeping your code, security, and architecture as simple as possible is a viable solution. Simpler systems are easier for identifying potential issues, maintenance, and applying additional security features. |
| 1. Default Deny | By default, all functions and users should be blocked from accessing data, resources, and functionalities that compromise sensitive information or the entire system. Defaulting to deny is a great principle to help mitigate the risk of a potential attack by ensuring that only trusted users and validating processes are granted permissions. Denying access by default reduces the risk of unauthorized usage of the system and prevents potential explorations from malicious attackers. |
| 1. Adhere to the Principle of Least Privilege | In addition to default deny, the Principle of Least Privilege is an integral step to securing your platform. This principle and security policy dictates that all users, applications, resources, and systems should be given the lowest level of access or permissions necessary to perform their given tasks. By utilizing this principle, you are ensuring that every process of your system is only able to complete its needed tasks and is unable to access other functions that could compromise the entirety of your system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems entails ‘cleaning’ and stripping away data to properly format information to prevent security vulnerabilities and data leakage. The process of sanitizing data includes removing special characters or scripts that could be used to exploit other parts of a larger system like databases and servers. Both the sending and receiving systems benefit greatly from sanitizing data. |
| 1. Practice Defense in Depth | Defense in Depth (DiD) is a security principle that involves implementing multiple layers of security, encapsulation, and obfuscation of information to protect against threats and potential malicious attacks. To simply visualize, Defense in Depth is like an onion. At the middle of the onion belongs the sensitive information, protected processes, and more. The layers around the middle of the onion serve as the layers of defense that shield the middle from the outside elements. Basic Defense in Depth examples that are already common are firewalls, encryption and 2FA authentication. Furthermore, DiD may serve as a ‘castle wall’ in how one layer may be compromised, but the other layers are still there to protect and defend the entirety of the system. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance is an effective security technique that is essential for identifying and mitigating security vulnerabilities earlier in the overall development process. For example, some common quality assurance practices include code reviews, automated testing, and penetration testing. These QA practices and techniques are used to catch potential flaws internally before deployment that could result in a drastic error once deployed publicly. |
| 1. Adopt a Secure Coding Standard | To conclude, adopting a secure coding standard provides a developer and a development team with a set of guidelines, rules and best practices that are proven to minimize security risks. Keeping these guidelines and rules in mind early, during, and after development can assist developers in avoiding common vulnerabilities and attacks such as buffer overflows, SQL injections, and improper error handling. Adopting a secure coding standard is a sure way for developers to be more conscious of security during the development process. |

### 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-DAT] | Using the appropriate data types guarantees that variables and inputs are limited to the developer's intended ranges and formats. This standard lowers the possibility of buffer underflow, overflow, and other incorrect data interpretation. By ensuring that system activities are operating as anticipated, data type validation reduces the possibility of a malicious attack. |

| **Noncompliant Code** |
| --- |
| The following example uses a buffer overflow or underflow to may cause out-of-bounds access. The security risks start by not validating the input size. |
| #include <iostream> using namespace std;  void processData(int input) {  char buffer[10];  buffer[input] = 'A';  *// no validation for input given* }  int main () {  int input;  std::cin >> input;  processData(input);  return 0; } |

| **Compliant Code** |
| --- |
| For the compliant example below, the input is validated to ensure that it falls within bounds and within the valid index range of the buffer. The following code prevents out-of-bounds access and ensures that the program handles user input correctly. |
| #include <iostream> using namespace std;  void processData(int input) {  char buffer[10];  if (input >= 0 && input < 10) {  buffer[input] = 'A';  } else {  cout << "Invalid input! Enter a number between 0 and 9." << endl;  } } |

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

| **Principles(s):**   * **Validate Input Data:** This principle supports the standard by ensuring that input values provided by users and other external sources are properly checked. * **Architect and Design for Security:** This principle emphasizes the importance of addressing vulnerabilities like buffer overflows earlier in the software development lifecycle. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.7 | Data type checker | This tool detects issues related to improper data type usage |
| SonarQube | 8.9 | CWE-119, S5146 | This tool detects buffer overflows and improper data validation |
| Astree | 20.04 | Memory access checker | This tool detects runtime memory errors such as buffer overflows caused by improper data handling and out-of-bounds-access |
| Helix QAC | 2023.1 | General Memory Check | This tool enforces correct data type usage and prevents memory access issues |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DAT] | Validating data value ensures that values assigned to variables and/or passed to different functions meet the specific criteria and format defined by the developer. This is important as this prevents the system from accepting and going through with internal processes, which may lead to errors and serious vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example below, the code does make a check to see if the input is greater than zero, but it doesn’t also check if the age falls within a realistic range. This means that an age of 50,000 would be allowed to pass and can lead to problematic behavior from the program. |
| #include <iostream> using namespace std;  void processData (int age) {  if (age > 0) {  std::cout << "Valid Age" << endl;  } else {  std::cout << "Invalid Age" << endl;  } }  int main() {  int input;  std::cin >> input;  processData(input);  return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, the input is validated to check for positive numbers and to fall within a realistic range. This validation ensures that the age entered is correct and protects against unwanted behavior. |
| #include <iostream> using namespace std;  void processData(int age) {  if (age > 0 && age <= 150) {  std::cout << "Valid age." << endl;  } else {  cout << "Invalid age. Enter a value between 1 and 150." << endl;  } }  int main() {  int input;  cin >> input;  processData(input);   return 0; } |

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

| **Principles(s):**   * **Validate Input Data:** This principle ensures that all input values are checked for accuracy and correctness before being processed. By validating input data, you are ensuring that the values passed are realistic and not potentially malicious. * **Adopt a Secure Coding Standard:** Using defined coding standards that enforce validation ensures the security of the system by eliminating and mitigating the risks of known vulnerabilities caused by improper data handling |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.04 | Data integrity checks | Astree performs static analysis to check for invalid data values that may cause runtime issues. |
| Helix QAC | 2024 | Input range validation | Validates input ranges and ensures that data values passed to functions conform to proper standards |
| SonarSource | 8.9 | CWE-704 | This tool detects missing input validation by checking data types and ranges. |
| Cppcheck | 2.7 | Data validation checker | This tool detects issues related to improper or missing validation of input data, ensuring compliance with the expected value ranges. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | Ensuring string correctness is a critical aspect of your program, as it prevents issues such as buffer overflows, injection attacks, and more. As with the other standards shown above, strings must be validated as well to check for length, format and string content to avoid accepting strings that are incorrect or strings that could compromise security of the system. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code below, I utilize the ‘strcpy’ function to copy the input of the string into a buffer size of 10 without checking for the length of the given input. This example could result in buffer overflows or other serious vulnerabilities. |
| #include <iostream> #include <cstring> using namespace std;  void processString(char\* input) {  char buffer[10];  strcpy(buffer, input);  }  int main() {  char input[100];  std::cin >> input;  processString(input);   return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, the input string’s length is validated to ensure the string size does not exceed the given buffer size. This example prevents buffer overflows and ensures that strings are being handled properly. |
| #include <iostream> #include <cstring> using namespace std;  void processString(const char\* input) {  char buffer[10];  if (strlen(input) < sizeof(buffer)) {   strcpy(buffer, input);  } else {  cout << "Input too long. Please enter a string with less than 10 characters." << endl;  } }  int main() {  char input[100];  cin >> input;  processString(input);  return 0; } |

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

| **Principles(s):**   * **Validate Input Data:** Validating string input prevents common vulnerabilities like buffer overflows and ensures that the string size is within safe bounds which reduces the risk of memory corruption or severe security breaches. * **Sanitize Data Sent to Other Systems:** This principle applies because unsafe strings can be used for injection attacks. By validating and sanitizing string inputs, the system is protected from potential attacks that exploit unsafe string handling and mitigates the risks of these unsafe strings. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.04 | Buffer and string checks | Astree detects buffer overflows and improper string handling that may result in unsafe memory operations and functions during runtime. |
| SonarSource | 8.9 | CWE-120 | This tool detects unsafe string handling that could lead to buffer overflows and formatted string exploits. |
| Cppcheck | 2.7 | String buffer checker | This tool identifies unsafe string operations, such as copying strings without validating their length. |
| CWE | CWE-120 | buffer Copy without size check | CWE-120 addresses vulnerabilities caused by unsafe string and buffer handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | SQL Injection usually occurs when user input isn’t validated accordingly and attackers modify input queries to access or manipulate the system. Properly handling user input is integral to preventing SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example below, the user input is directly inserted in the SQL query string, which is dangerous as it allows a malicious user to inject SQL commands directly. |
| #include <iostream> #include <string> using namespace std;  void getUserData(const string& username) {  string query = "SELECT \* FROM users WHERE username = '" + username + "';";  cout << "Executing query: " << query << endl; *// User input is directly concatenated into the SQL query* }  int main() {  string username;  cout << "Enter username: ";  cin >> username;  getUserData(username); *// Potentially dangerous query execution*  return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, a query string is treated as a parameter instead of directly inserting it as query string. |
| #include <iostream> using namespace std;  void getUserData(const string& username) {  string query = "SELECT \* FROM users WHERE username = ?";  cout << "Executing prepared query: " << query << " with parameter: " << username << endl;  *// User input is treated as a parameter, preventing SQL injection* }  int main() {  string username;  cout << "Enter username: ";  cin >> username;  getUserData(username);   return 0; } |

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

| **Principles(s):**   * **Sanitize Data Sent to Other Systems:** This principle focuses on sanitizing and validating any data before it interacts with a database. SQL injection attacks usually occur due to improper input sanitization. * **Adopt a Secure Coding Standard:** Following trusted practices such as utilizing parameterized queries ensures that the risk of SQL injection attacks are mitigated and prevented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Input Validation Checks | This tool analyzes code to ensure user input is properly validated and sanitized. |
| SonarSource | 2024 | RSPEC-2077 | Detects unsafe SQL queries that are vulnerable to SQL injections by analyzing input handling. |
| Cppcheck | 2.9 | SQL injection checker | Detects possible SQL injection vulnerabilities by checking for user inputs that are directly used in SQL queries |
| CWE | CWE-89 | SQL Injection | CWE-89 refers to improper handling of input in SQL queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | Proper memory protection prevents dangerous vulnerabilities such as buffer overflows and memory leaks. By validating memory accesses, the system is more secure from malicious attacks focusing on memory access. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example below, memory is allocated for an array of 10 integers but the program accesses the index at 15, which leads to unwanted behavior and potentially insecure software. |
| #include <iostream> using namespace std;  void processData() {  int\* array = new int[10];  array[15] = 100; *// Out-of-bounds access*  delete[] array;  }  int main() {  processData(); *// Unsafe memory access*  return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, the solution is checking bounds before attempting to access the memory. |
| #include <iostream> using namespace std;  void processData() {  int\* array = new int[10];  if (array != nullptr && 15 < 10) { *// bounds checking to ensure safe access*  array[15] = 100;  } else {  std::cout << "Invalid memory access attempt!" << endl;  }  delete[] array; *// Properly deallocate memory* }  int main() {  processData(); *// Safe memory handling*  return 0; } |

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

| **Principles(s):**   * **Validate Input Data:** Validating memory addresses and bounds ensures that input values are within the expected ranges, which prevents out-of-bounds memory access. * **Heed Compiler Warnings:** Many memory related issues such as buffer overflows can be caught by compiler warnings. * **Adopt a Secure Coding Standard:** Implementing coding standards such as checking pointers and ensuring that memory is freed accordingly is paramount to reducing the risk of memory management vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Memory Safety Checks | This tool performs static analysis to detect buffer overflows and other memory related safety issues |
| SonarSource | 2024 | RSPEC-3735 | This tool identifies memory leak vulnerabilities when memory is freed improperly. |
| Cppcheck | 2.9 | Memory Leak Checker | This tool detects memory leaks and improper memory deallocation practices |
| CWE | CWE-415 | Memory Leak Checker | CWE-415 refers to vulnerabilities caused by freeing memory or using it after the memory has been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-AST] | Assertions are a useful integration that verifies assumptions in code during development and during testing phases. These assertions are used to help catch logical errors and unexpected behavior throughout your code. Assertions are mostly used during development and should be avoided when pushing to production as they can lead to security vulnerabilities and unexpected behaviors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example below, an assertion is used to check if the input given is non-negative. |
| #include <iostream> #include <cassert> using namespace std;  void processInput(int value) {  assert(value >= 0); *// Using an assertion to validate input*  std::cout << "Processing value: " << value << endl; }  int main() {  int input;  std::cin >> input;  processInput(input); *// If input is negative, behavior is undefined*  return 0; } |

| **Compliant Code** |
| --- |
| In this compliant code example, assertions are removed for production release to ensure undefined behavior is prevented. |
| #include <iostream> using namespace std;  void processInput(int value) {  if (value >= 0) { *// runtime check for input validation*  std::cout << "Processing value: " << value << endl;  } else {  std::cout << "Invalid input. Value must be non-negative." << endl;  } }  int main() {  int input;  std::cin >> input;  processInput(input); *// Input is validated at runtime*  return 0; } |

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

| **Principles(s):**   * **Validate Input Data:** Assertions should never replace proper input validation in final production code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Assertion misuse check | This tool performs static analysis to ensure that assertions are not used in production code |
| SonarSource | 2024 | RSPEC-4541 | This tool detects assertions in production environments where they may introduce unintended behaviors or vulnerabilities |
| Cppcheck | 2.9 | Assertion Checker | This tool identifies improper use of assertions in production code |
| CWE | CWE-617 | Assertion Used in production | CWE-617 addresses the risks of assertions in production code |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | Exceptions are used to handle runtime errors and unexpected behavior. Although exceptions are useful and essential when used appropriately, improper use of exceptions can lead to severe bugs, security vulnerabilities and an unstable system overall. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a string is thrown as an exception, which is not recommended. Also, the try-catch blocks catches any exception without specifically declaring which type. |
| #include <iostream> using namespace std;  void processInput(int value) {  try {  if (value < 0) {  throw "Negative value error"; *// Throwing a string as an exception*  }  cout << "Processing value: " << value << endl;  } catch (...) {  cout << "An error occurred." << endl; *// Catching all exceptions without proper handling*  } }  int main() {  int input;  cin >> input;  processInput(input);  return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, a specific exception type is thrown and caught. This makes it easier to debug any errors that may rise since the developer can handle the exception in a more predictable manner. |
| #include <iostream> #include <stdexcept> using namespace std;  void processInput(int value) {  try {  if (value < 0) {  throw invalid\_argument("Negative value error"); *// Throwing a proper exception*  }  cout << "Processing value: " << value << endl;  } catch (const invalid\_argument& e) {  cout << "Error: " << e.what() << endl; *// Catching specific exception type*  } }  int main() {  int input;  cin >> input;  processInput(input); *// Input is processed with appropriate exception handling*  return 0; } |

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

| **Principles(s):**   * **Adopt a Secure Coding Standard:** This principle enforces the need for structured exception handling and encourages developers to use specific exception types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Exception handling checks | This tool analyzes code to ensure that exception are handled properly, avoiding catch blocks that may mask and worsen underlying errors and vulnerabilities.v |
| SonarSource | 2024 | RSPEC-1440 | Detects overly general exception handling and encourages improving code clarity |
| Cppcheck | 2.9 | Exception handling checker | Identifies general catch blocks and improper use of exceptions |
| CWE | CWE-396 | Catching Generic Exceptions | CWE-396 refers to improper handling of exceptions that may result in incorrect error management. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Management | [STD-008-FIL] | Proper file management is essential to avoiding data leaks in your system, which can lead to poor performance, security vulnerabilities or system crashes. When opening a file, for example, it is important to handle the processes correctly and close the file at the end to avoid sensitive data from leaking. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a file is opened but not explicitly closed after use. This is dangerous as it can result in a resource leak. |
| #include <iostream> #include <fstream> using namespace std;  void readFile(const string& filename) {  ifstream file;  file.open(filename); *// file resource opened but not properly closed*  if (!file) {  cout << "Failed to open file." << endl;  return;  }    *// Forgetting to close the file*  }  int main() {  readFile("data.txt"); *// resource leak due to unclosed file*  return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, the file is closed after use which ensures that the file resource is properly released out of the memory and avoids leaks. |
| #include <iostream> #include <fstream> using namespace std;  void readFile(const string& filename) {  ifstream file;  file.open(filename);  if (!file) {  cout << "Failed to open file." << endl;  return;  }    file.close(); *// Properly closing the resource after use* }  int main() {  readFile("data.txt"); *// resource is safely managed*  return 0; } |

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

| **Principles(s):**   * **Validate Input Data:** Input data should be checked for type, length, format, and range before being accepted by the system. This should prevent vulnerabilities such as buffer overflows and injection attacks * **Sanitize Data Sent to Other Systems:** Input Data should be sanitized and escaped properly before sending it to any other subsystem such as the database. This ensures that the untrusted input cannot be used for injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Input Handling Checks | This tool analyzes code to ensure that user input is validated, sanitized and handled properly before being used internally by any subsystem |
| SonarSource | 2024 | RSPEC-5167 | Detects improper input validation and sanitization, ensuring that the input is properly validated before being processed by the system. |
| Cppcheck | 2.9 | Input validation checker | Identifies improper input validation practices, ensuring all inputs are properly sanitized and validated for expected length and type |
| CWE | CWE-20 | Improper Input Validation | CWE-20 refers to improper input validation, which can lead to buffer overflows, injection attacks, or other unexpected behaviors that can lead to severe vulnerabilities |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | [STD-009-ERR] | Proper error handling ensures that failures, logical errors, and unexpected behaviors are managed properly. Failing to handle errors can result in security vulnerabilities or data loss. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, there is no check that ensures the denominator is non-zero before performing the division. |
| #include <iostream> using namespace std;  void divide(int a, int b) {  int result = a / b; *// No check for division by zero*  cout << "Result: " << result << endl; }  int main() {  divide(10, 0); *// Program crashes due to division by zero*  return 0; } |

| **Compliant Code** |
| --- |
| In the compliant code example below, a check is performed to ensure that division by zero does not occur. The error is handled properly and ends the program. |
| #include <iostream> using namespace std;  void divide(int a, int b) {  if (b != 0) {  int result = a / b;  cout << "Result: " << result << endl;  } else {  cout << "Error: Division by zero is not allowed." << endl; *// Handling error*  } }  int main() {  divide(10, 0); *// Error is handled*  return 0; } |

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

| **Principles(s):**   * Encrypt Data: Encryption should be applied to sensitive information to ensure its confidentiality and security. Strong encryption algorithms must be used to protect against unauthorized access. * Adopt a Secure Coding Standard: Secure coding practices should include proper encryption to protect sensitive data from being exposed or compromised. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Encryption usage checks | Astree ensures that sensitive data is encrypted accordingly and flags cases where unencrypted sensitive data is stored or transmitted across the system. |
| SonarSource | 2024 | RSPEC-4426 | Detects cases where sensitive data is sent or stored without encryption. |
| Cppcheck | 2.9 | Encryption Checker | Identifies missing encryption in areas where sensitive data is handled or stored. |
| CWE | CWE-311 | Missing Encryption of Sensitive Data | CWE-311 refers to the failure to encrypt sensitive data which could result in unauthorized access and exposed sensitive data. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Logging | [STD-010-LOG] | Secure loggins ensure that system events and processes are recorded properly without exposing sensitive information, such as password, financial information, and more. By filtering log entries, sensitive data is kept secure and reduces the risks of a data leakage. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example below, both the username and password are logged directly into a file. This can lead to sensitive data leakage and system breaches. Sensitive information should never be logged internally. |
| #include <iostream> #include <fstream> #include <string> using namespace std;  void logEvent(const string& message) {  *// Logging sensitive data directly to a log file*  ofstream logFile("application.log", ios\_base::app);  logFile << message << endl;  logFile.close(); }  void processLogin(const string& username, const string& password) {  *// Log both username and password, which is unsafe*  logEvent("Login attempt by user: " + username + " with password: " + password);    *// Simulate login process*  if (username == "admin" && password == "SuperSecret123") {  logEvent("User: " + username + " successfully logged in.");  } else {  logEvent("User: " + username + " failed to log in.");  } }  int main() {  string username = "admin";  string password = "SuperSecret123";    processLogin(username, password); *// Sensitive data is exposed in logs*  return 0; |

| **Compliant Code** |
| --- |
| In the compliant code example below, sensitive information such as the password are not logged to ensure that no sensitive information is exposed in the logs. |
| #include <iostream> #include <fstream> #include <string> #include <ctime> using namespace std;  *// secure logging function with no sensitive data* void logEvent(const string& message) {  ofstream logFile("secure\_application.log");   *// add timestamp to log entries for better tracking*  time\_t now = time(0);  char\* dt = ctime(&now);  logFile << "[" << dt << "] " << message << endl;    logFile.close(); }  void processLogin(const string& username, const string& ipAddress) {  *//log only non-sensitive data like username and IP address*  logEvent("Login attempt by user: " + username + " from IP: " + ipAddress);   *// simulate login process*  bool success = (username == "admin");    if (success) {  logEvent("User: " + username + " successfully logged in.");  } else {  logEvent("User: " + username + " failed to log in.");  } }  int main() {  string username = "admin";  string ipAddress = "192.168.1.100";     processLogin(username, ipAddress);  return 0; } |

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

| **Principles(s):**   * Handle Errors: This principle ensures that errors are handled securely and all errors are handled accordingly |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.1 | Error handling checks | Astree checks for secure error handling practices, ensuring that specific exceptions are handled and generic catch-all blocks are avoided |
| SonarSource | 2024 | RSPEC-3984 | Detects improper error handling patterns |
| Cppcheck | 2.9 | Error handling checker | Detects improper use of error handling mechanisms |
| CWE | CWE-248 | Uncaught Exception | CWE-248 refers to cases where exceptions are not properly handled, which could lead to potential system crashes and other security vulernabilities |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-DAT | High | Likely | Medium | High | 3 |
| STD-002-DAT | Medium | Possible | Low | Medium | 2 |
| STD-003-STR | High | Likely | Medium | High | 3 |
| STD-004-SQL | High | Likely | Medium | High | 4 |
| STD-005-MEM | High | Likely | Medium | High | 4 |
| STD-006-AST | Medium | Likely | Low | Medium | 2 |
| STD-007-EXC | High | Likely | Medium | High | 3 |
| STD-008-FIL | High | Likely | Medium | High | 4 |
| STD-009-ERR | High | Likely | Medium | High | 4 |
| STD-010-LOG | Medium | Likely | Medium | High | 3w |

### 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 | Data stored on disks or databases must be encrypted using encryption methods such as AES-256 to prevent unauthorized access |
| Encryption in flight | All data transmitted over networks must be encrypted using common protocols such as TLS or HTTPS to avoid interception from malicious agents |
| Encryption in use | Data actively being processed in memory should be encrypted and handled securely |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Ensure strong multi-factor authentication for all users accessing the system and subsystems |
| Authorization | Implement role-based access control to limit user access to only what is necessary for their role and tasks needed to be completed |
| Accounting | Maintain properly formatted audit logs of all user activities, including logins, file accesses, or user permission changes. |

**\***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 | 10/12/2024 | Milestone | Steven Rodas | N/A |
| 3.0 | 10/18/2024 | Project One | Steven Rodas | N/A |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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