

# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.7kc3ovld5gu9)

[Purpose 2](#_heading=h.ivhi7abvid7a)

[Scope 2](#_heading=h.p3xtraygs4ex)

[Module Three Milestone 2](#_heading=h.w26sjmnwkfy4)

[Ten Core Security Principles 2](#_heading=h.x6n0od6froh6)

[C/C++ Ten Coding Standards 3](#_heading=h.wfr26sbhi0tm)

[Coding Standard 1 4](#_heading=h.whvz92zcc56v)

[Coding Standard 2 5](#_heading=h.14zx2ed2iuho)

[Coding Standard 3 6](#_heading=h.uwrpzffsnz3n)

[Coding Standard 4 7](#_heading=h.fvge42egsf3)

[Coding Standard 5 8](#_heading=h.tb2fa1hj2byl)

[Coding Standard 6 9](#_heading=h.xoofxp4dpak1)

[Coding Standard 7 10](#_heading=h.k3tz3if11qxk)

[Coding Standard 8 11](#_heading=h.xcx8t7f8xyha)

[Coding Standard 9 13](#_heading=h.5m8k4o2r9uh)

[Coding Standard 10 14](#_heading=h.ox1o6zq3bg7p)

[Defense-in-Depth Illustration 15](#_heading=h.cj8xnmis776r)

[Project One 15](#_heading=h.x0edcypcvoql)

[1.](#_heading=h.5ygiuw7oqahj) Revise the C/C++ Standards 15

[2.](#_heading=h.6qzht49ie9o5) Risk Assessment 15

[3.](#_heading=h.826yvl6q8ra3) Automated Detection 15

[4.](#_heading=h.eswo5vx1facn) Automation 15

[5.](#_heading=h.f42w3mm6qgxy) Summary of Risk Assessments 16

[6.](#_heading=h.qwckho9fdrf6) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.ccq3nigtq3va) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ivfr47j5m9x)

[Enforcement 18](#_heading=h.xo98opywnqqa)

[Exceptions Process 18](#_heading=h.dgrklyljpaxn)

[Distribution 19](#_heading=h.sziqjl20ekqd)

[Policy Change Control 19](#_heading=h.c9y9lq7otzys)

[Policy Version History 19](#_heading=h.tm7mlqjad4ac)

[Appendix A Lookups 19](#_heading=h.rzzrs53frdtn)

[Approved C/C++ Language Acronyms 19](#_heading=h.mrdb3e9qxgd)

## 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 | All external and internal inputs must be validated to prevent malicious data from causing unexpected behavior. This minimizes the risk of injection attacks and data corruption. |
| 1. Heed Compiler Warnings | Compiler warnings are early indicators of potential coding issues; addressing them proactively reduces vulnerabilities that could be exploited. |
| 1. Architect and Design for Security Policies | Security must be built into the design phase. Systems should be planned with layered defenses and threat models to reduce the likelihood of compromise. |
| 1. Keep It Simple | Simplicity in design and coding minimizes complexity, reducing the risk of errors that may lead to security weaknesses. |
| 1. Default Deny | Systems should deny access or assume the least privilege by default, only granting access when explicitly necessary. |
| 1. Adhere to the Principle of Least Privilege | Users and processes must operate with the minimum privileges required for functionality, reducing the potential impact of security breaches. |
| 1. Sanitize Data Sent to Other Systems | Data leaving a system must be scrutinized and sanitized to ensure it does not introduce vulnerabilities in interconnected systems. |
| 1. Practice Defense in Depth | Multiple security controls should be employed at various layers so that if one fails, others still provide protection. |
| 1. Use Effective Quality Assurance Techniques | A combination of unit testing, integration testing, and vulnerability scanning must be implemented continuously to catch and remediate defects early. |
| 1. Adopt a Secure Coding Standard | Adhering to established secure coding guidelines (e.g., SEI CERT) ensures consistency and reduces the risk of vulnerabilities across codebases. |

### 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 Integrity** | STD-001-CPP | Consistent use of data types prevents errors from implicit type conversions and avoids buffer overflows or memory misinterpretations. |

| **Noncompliant Code** |
| --- |
| Implicit conversion of an integer to a smaller type without explicit casting, leading to overflow. |
| #include <iostream>  int main() {  long bigNumber = 100000L;  short smallNumber = bigNumber;  std::cout << "Small number: " << smallNumber << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using explicit type casts and static assertions to ensure that data types remain consistent. |
| #include <iostream>  #include <limits>  #include <stdexcept>  int main() {  long bigNumber = 100000L;  if (bigNumber > std::numeric\_limits<short>::max() ||  bigNumber < std::numeric\_limits<short>::min()) {  throw std::overflow\_error("bigNumber cannot be represented as a short.");  }  short smallNumber = static\_cast<short>(bigNumber);  std::cout << "Small number: " << smallNumber << std::endl;  return 0;  } |

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

| **Principles(s):** This standard ensures that variables are declared with the correct data types to prevent implicit conversions that might lead to overflow or loss of precision. It maps to:  Validate Input Data (Principle 1): Ensuring that data stays within expected types.  Keep It Simple (Principle 4): Using explicit and clear type declarations to avoid hidden errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | “Data\_Type\_Integrity” rule | Scans code for implicit type conversions that could lead to memory corruption. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value Validation** | STD-002-CPP | Ensures that data values are within expected ranges and formats before processing to prevent logic errors and misuse. |

| **Noncompliant Code** |
| --- |
| Failing to check numeric inputs for overflows before arithmetic operations. |
| #include <iostream>  int main() {  int percentage;  std::cout << "Enter a percentage (0-100): ";  std::cin >> percentage;  // No verification that input is within the valid range.  std::cout << "You entered: " << percentage << "%" << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Implementing bounds checks and using safe arithmetic libraries. |
| #include <iostream>  #include <limits>  int main() {  int percentage;  std::cout << "Enter a percentage (0-100): ";  std::cin >> percentage;  if (std::cin.fail() || percentage < 0 || percentage > 100) {  std::cerr << "Invalid input. Please enter a value between 0 and 100." << std::endl;  // Optionally, clear the error and ignore the rest of the line  std::cin.clear();  std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  return 1;  }  std::cout << "You entered: " << percentage << "%" << std::endl;  return 0;  } |

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

| **Principles(s):** This standard focuses on validating the values of data before they are processed. It ensures inputs remain within expected boundaries, which is critical to preventing logic errors or potential vulnerabilities. This approach maps to:  Validate Input Data (Principle 1)  Adopt a Secure Coding Standard (Principle 10) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 2021.3 | “Data\_Validation\_Rule” | Monitors arithmetic operations and data assignments for out-of-range values. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Preventing off-by-one errors, buffer overflows, and ensuring proper string termination protect against many common vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Using unsafe string functions such as strcpy without verifying buffer sizes. |
| #include <cstring>  #include <iostream>  int main() {  char src[10] = "Hello";  char dest[5];  strcpy(dest, src);  std::cout << "Destination: " << dest << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Utilizing secure functions like strncpy\_s with explicit length parameters. |
| #include <iostream>  #include <string>  int main() {  std::string src = "Hello";  std::string dest = src;  std::cout << "Destination: " << dest << std::endl;  return 0;  } |

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

| **Principles(s):** Ensuring proper handling of strings prevents common issues like buffer overflows or off-by-one errors. By using safe string functions or C++ string objects, you reduce the risk of memory-related vulnerabilities. This maps to:  Validate Input Data (Principle 1)  Practice Defense in Depth (Principle 8) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.1 | “String\_Safety\_Check” | Detects the use of unsafe string manipulation functions. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Avoiding concatenation of SQL queries with untrusted input prevents one of the most widespread vulnerabilities in software applications. |

| **Noncompliant Code** |
| --- |
| Building SQL queries by concatenating user-supplied input with query strings. |
| #include <iostream>  #include <string>  void unsafeQuery(const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  std::cout << "Executing query: " << query << std::endl;  }  int main() {  std::string input = "admin' OR '1'='1"; // Malicious input  unsafeQuery(input);  return 0;  } |

| **Compliant Code** |
| --- |
| Employing parameterized queries or prepared statements to bind inputs safely. |
| #include <iostream>  #include <mysql\_driver.h>  #include <mysql\_connection.h>  #include <cppconn/prepared\_statement.h>  void safeQuery(const std::string& userInput) {  sql::mysql::MySQL\_Driver \*driver = sql::mysql::get\_mysql\_driver\_instance();  std::unique\_ptr<sql::Connection> con(driver->connect("tcp://127.0.0.1:3306", "user", "password"));  con->setSchema("database");  std::unique\_ptr<sql::PreparedStatement> pstmt(  con->prepareStatement("SELECT \* FROM users WHERE username = ?")  );  pstmt->setString(1, userInput);  std::cout << "Executing safe prepared statement query." << std::endl;  std::unique\_ptr<sql::ResultSet> res(pstmt->executeQuery());  }  int main() {  std::string input = "admin";  safeQuery(input);  return 0;  } |

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

| **Principles(s):** SQL injection can be mitigated by avoiding string concatenation of untrusted data into SQL commands. Instead, using parameterized queries or prepared statements ensures that user input is properly escaped, mapping to:  Validate Input Data (Principle 1)  Sanitize Data Sent to Other Systems (Principle 7) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.8 | “SQL\_Injection\_Rule” | Flags dynamically built SQL queries with embedded user inputs. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Proper handling of memory allocation and deallocation minimizes risks such as memory leaks and buffer overflows, which can lead to remote code execution. |

| **Noncompliant Code** |
| --- |
| Using raw pointers without bounds checking and manual memory management. |
| #include <iostream>  void riskyFunction() {  int\* data = new int[100];  // Some processing...  // Exception or early return might skip delete[]  delete[] data;  }  int main() {  riskyFunction();  return 0;  } |

| **Compliant Code** |
| --- |
| Using smart pointers and bounds-checked array accesses in C++. |
| #include <iostream>  #include <vector>  void safeFunction() {  std::vector<int> data(100);  // Perform processing on data; memory automatically managed.  }  int main() {  safeFunction();  return 0;  } |

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

| **Principles(s):**Safe memory management is crucial to avoid buffer overflows, leaks, or dangling pointers. The use of smart pointers and containers aligns with:  Keep It Simple (Principle 4)  Practice Defense in Depth (Principle 8) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | “Memory\_Leak\_Check” | Analyzes dynamic memory allocation and verifies safe usage patterns. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions help detect logic errors during development by verifying that critical conditions hold true at runtime. |

| **Noncompliant Code** |
| --- |
| Omitting assertions in areas where invariants are assumed, increasing the risk of undetected faults. |
| #include <iostream>  int divide(int a, int b) {  return a / b;  }  int main() {  std::cout << "Result: " << divide(10, 0) << std::endl; // Runtime error likely.  return 0;  } |

| **Compliant Code** |
| --- |
| Employing assertions to verify inputs, states, and post-conditions where appropriate. |
| #include <iostream>  #include <cassert>  int divide(int a, int b) {  // Using an assertion to ensure that b is not zero.  assert(b != 0 && "Divider cannot be zero.");  return a / b;  }  int main() {  int x = 10, y = 2; // Modify y to a nonzero value to pass the assertion.  std::cout << "Result: " << divide(x, y) << std::endl;  return 0;  } |

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

| **Principles(s):** Assertions help catch unexpected conditions during development. Their use aids in validating critical assumptions and maps to:  Use Effective Quality Assurance Techniques (Principle 9)  Adopt a Secure Coding Standard (Principle 10) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Custom CI/CD Test Framework | Internal Build 3.1 | “Assertion\_Check” | Ensures that critical assertions are present and active within unit tests. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling is crucial to prevent application crashes and sensitive data leaks that can result from unhandled exceptions. |

| **Noncompliant Code** |
| --- |
| Catching all exceptions generically and failing to handle cleanup. |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  try {  throw std::runtime\_error("Unexpected error");  } catch (...) {  std::cerr << "An error occurred." << std::endl;  }  }  int main() {  riskyOperation();  return 0;  } |

| **Compliant Code** |
| --- |
| Using specific exception types and ensuring proper resource deallocation in exception blocks. |
| #include <iostream>  #include <stdexcept>  void robustOperation() {  try {  throw std::runtime\_error("Detailed error message");  } catch (const std::runtime\_error& ex) {  std::cerr << "Runtime error: " << ex.what() << std::endl;  } catch (const std::exception& ex) {  std::cerr << "Error: " << ex.what() << std::endl;  }  }  int main() {  robustOperation();  return 0;  } |

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

| **Principles(s):** Robust exception handling ensures that errors are caught and processed properly rather than allowing the application to crash. This standard maps to:  Practice Defense in Depth (Principle 8)  Adopt a Secure Coding Standard (Principle 10) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 13.0 | “Exception\_Safety\_Check” | Checks for generic exception catches and recommends more granular handling. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | Enforcing bounds on buffer operations is essential to avoid overflows that compromise application memory. |

| **Noncompliant Code** |
| --- |
| Failing to validate indices or using unsafe functions like gets(). |
| #include <cstdio>  #include <cstring>  int main() {  char buffer[10];  gets(buffer); // Deprecated and dangerous.  printf("Buffer: %s\n", buffer);  return 0;  } |

| **Compliant Code** |
| --- |
| Using safe functions and validating buffer sizes before operations. |
| #include <iostream>  #include <limits>  #include <string>  int main() {  std::string input;  std::cout << "Enter text (max 9 characters): ";  std::getline(std::cin, input);  if (input.size() > 9) {  std::cerr << "Input too long!" << std::endl;  return 1;  }  char buffer[10];  strncpy(buffer, input.c\_str(), sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0';  std::cout << "Buffer: " << buffer << std::endl;  return 0;  } |

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

| **Principles(s):** This standard is focused on verifying that all buffers are handled within their bounds to prevent overwrites. It maps to:  Keep It Simple (Principle 4)  Practice Defense in Depth (Principle 8) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 20.1 | “Buffer\_Overflow\_Check” | Analyzes code for potential overruns in array or pointer operations. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-009-CPP | Ensuring that resources (memory, file handles, network sockets) are properly allocated and released prevents leaks and maintains application performance and security. |

| **Noncompliant Code** |
| --- |
| Not releasing file descriptors or memory in error paths. |
| #include <cstdio>  #include <iostream>  void readFile(const char\* filename) {  FILE\* file = fopen(filename, "r");  if (!file) {  std::cerr << "Unable to open file!" << std::endl;  return;  }  // File is processed...  fclose(file);  }  int main() {  readFile("data.txt");  return 0;  } |

| **Compliant Code** |
| --- |
| Implementing RAII (Resource Acquisition Is Initialization) to ensure automatic cleanup. |
| #include <iostream>  #include <fstream>  #include <string>  void readFile(const std::string& filename) {  std::ifstream file(filename);  if (!file.is\_open()) {  std::cerr << "Unable to open file!" << std::endl;  return;  }  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }  int main() {  readFile("data.txt");  return 0;  } |

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

| **Principles(s):** Proper management of system resources (e.g., memory, file handles) is paramount to prevent leaks. The RAII (Resource Acquisition Is Initialization) pattern is recommended. This maps to:  Keep It Simple (Principle 4)  Practice Defense in Depth (Principle 8) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17 | “Resource\_Leak\_Check” | Monitors resource allocation and verifies that cleanup is performed on all error paths. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging and Auditing | STD-010-CPP | Logging must be implemented securely so that sensitive data is not exposed and audit trails remain tamper-proof for forensic analysis. |

| **Noncompliant Code** |
| --- |
| Logging sensitive information (e.g., passwords, personal data) in plain text. |
| #include <iostream>  #include <fstream>  #include <string>  void logUserData(const std::string& username, const std::string& password) {  std::ofstream logFile("app.log", std::ios::app);  logFile << "User: " << username << ", Password: " << password << std::endl;  }  int main() {  logUserData("user123", "secretPass");  return 0;  } |

| **Compliant Code** |
| --- |
| Sanitizing log entries and storing logs in secure, access-controlled locations. |
| #include <iostream>  #include <fstream>  #include <string>  std::string maskPassword(const std::string& password) {  return std::string(password.length(), '\*');  }  void logUserData(const std::string& username, const std::string& password) {  std::ofstream logFile("app.log", std::ios::app);  logFile << "User: " << username  << ", Password: " << maskPassword(password) << std::endl;  }  int main() {  logUserData("user123", "secretPass");  return 0;  } |

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

| **Principles(s):** Logging must be implemented carefully so that sensitive data is not exposed in plaintext logs. Instead, log messages should be sanitized and access to log files restricted. This standard maps to:  Sanitize Data Sent to Other Systems (Principle 7)  Use Effective Quality Assurance Techniques (Principle 9) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ELK Stack | Custom Integration 1.0 | “Sensitive\_Data\_Logging\_Check” | Monitors logs for occurrences of sensitive information and enforces logging policies. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

Defense in Depth is a layered security strategy that protects systems and data by applying multiple overlapping controls at every stage of the software development lifecycle (SDLC). The illustration below conceptually represents how security layers work together to mitigate risks even if one layer is breached:

* Planning & Risk Analysis: Security is built into the planning phase through threat modeling, security metrics, and training. Automated risk assessment tools can help identify and prioritize vulnerabilities in the early stages.
* Secure Code Creation: Developers adhere to revised C/C++ coding standards, employing best practices such as explicit data type validation, secure string handling, parameterized queries to prevent SQL injection, and proper memory management. The illustration shows multiple checkpoints where secure coding standards are enforced using IDE plugins and version-controlled code repositories.
* Verification & Testing: Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), Interactive Application Security Testing (IAST), and Software Composition Analysis (SCA) work in concert to detect issues during build and integration. This phase is depicted as a layer that continuously scans code and alerts developers of noncompliance.
* Pre-Production and Release: Prior to deployment, integration tests, input fuzzing, chaos testing, and digital signature checks (software signing) ensure the code is robust and secure. The diagram marks this stage as a transitional layer where vulnerability remediation is enforced before code moves to production.
* Prevent & Detect in Production: In production environments, multiple real-time security controls are employed. Tools such as Runtime Application Self-Protection (RASP), User and Entity Behavior Analytics (UEBA), network monitoring, and access controls continuously protect the system. These layers are illustrated as protective barriers that detect anomalous behavior and enable rapid incident response.
* Incident Response & Adaptation: When security breaches occur, integrated security orchestration and automated incident response measures help mitigate damage and quickly adapt policies based on correlated threat intelligence. The final layer emphasizes continuous improvement through post-incident analysis and patch updates.

This illustration underscores that no single defensive measure is sufficient; instead, the security of the Green Pace environment is maintained by leveraging multiple, redundant controls throughout every phase of the SDLC.

### 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 | High | 2 |
| STD-003-CPP | High | Likely | Medium | High | 3 |
| STD-004-CPP | High | Likely | Medium | High | 3 |
| STD-005-CPP | High | Medium | High | High | 3 |
| STD-006-CPP | Medium | Medium | Low | Medium | 2 |
| STD-007-CPP | High | Medium | Medium | High | 3 |
| STD-008-CPP | High | Likely | High | High | 3 |
| STD-009-CPP | Medium | Medium | Medium | Medium | 2 |
| STD-010-CPP | Medium | Medium | Low | Medium | 2 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 | All sensitive data stored on disk or databases must be encrypted using AES‑256. This policy mitigates the risk of unauthorized data exposure if storage media are compromised. |
| Encryption in flight | Data transmitted across networks is required to use TLS 1.2 or higher to ensure confidentiality and integrity against man‑in‑the‑middle attacks. |
| Encryption in use | When processing sensitive data, in-memory encryption mechanisms should be used where available. This minimizes exposure during computation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All systems require strong multifactor authentication (MFA) to verify user identities prior to access. |
| Authorization | Role‑based access control (RBAC) is enforced to ensure users access only the information and functions necessary for their roles. |
| Accounting | Comprehensive audit logs record user actions including login attempts, file accesses, and changes to sensitive data. Logs are retained securely and reviewed regularly for anomalies. |

**\***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

| **Coding Standard** | **Mapped Principles** | **Explanation** |
| --- | --- | --- |
| STD-001-CPP | Principle 1: Validate Input Data; Principle 4: Keep It Simple | Clear data type declarations and explicit casts prevent erroneous type conversions. |
| STD-002-CPP | Principle 1: Validate Input Data; Principle 10: Adopt a Secure Coding Standard | Input validation ensures only acceptable values are processed, reducing logic errors. |
| STD-003-CPP | Principle 1: Validate Input Data; Principle 8: Practice Defense in Depth | Proper string handling prevents buffer overruns and unexpected behavior. |
| STD-004-CPP | Principle 1: Validate Input Data; Principle 7: Sanitize Data Sent to Other Systems | Parameterized queries eliminate risk by separating user input from executable code. |
| STD-005-CPP | Principle 4: Keep It Simple; Principle 8: Practice Defense in Depth | Consistent memory management minimizes the risk of buffer overflows and leaks. |
| STD-006-CPP | Principle 9: Use Effective Quality Assurance Techniques; Principle 10: Adopt a Secure Coding Standard | Assertions help catch potential errors early in the development lifecycle. |
| STD-007-CPP | Principle 8: Practice Defense in Depth; Principle 10: Adopt a Secure Coding Standard | Robust exception handling ensures that errors are caught and handled gracefully. |
| STD-008-CPP | Principle 4: Keep It Simple; Principle 8: Practice Defense in Depth | Validating buffer sizes and using safe functions prevents buffer overflow vulnerabilities. |
| STD-009-CPP | Principle 4: Keep It Simple; Principle 8: Practice Defense in Depth | Automated resource management via RAII ensures that resources are always appropriately released. |
| STD-010-CPP | Principle 7: Sanitize Data Sent to Other Systems; Principle 9: Use Effective Quality Assurance Techniques | Secure logging practices protect sensitive data from exposure while ensuring auditability. |

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 | 4/13/2025 | Added coding standards examples and encryption sections; revised Triple‑A policies and mapping principles; updated audit controls, enforcement, and exceptions process | Carson Denison | [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 |