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

**Andy Martinez**



# 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 validate input from untrusted sources to prevent injection vulnerabilities, data corruption, and potential security breaches. Use techniques such as input length checks, type validation, and format verification. Regular expressions and strict data validation functions help ensure input integrity. |
| 1. Heed Compiler Warnings | Always address and eliminate compiler warnings, as they can indicate potential vulnerabilities or undefined behaviors in the code. Treat warnings as errors and use compiler flags to enforce stricter checking. Regularly update the compiler to leverage improved analysis and warning capabilities. |
| 1. Architect and Design for Security Policies | Incorporate security considerations from the initial design phase to reduce vulnerabilities and ensure that the system is resilient against threats. Design with threat modeling and risk assessment practices to identify and mitigate potential weaknesses. |
| 1. Keep It Simple | Avoid overly complex code and architectures that can introduce unintended security flaws. Simplicity enhances maintainability and reduces the chances of introducing subtle bugs that are hard to detect and exploit. |
| 1. Default Deny | Follow the principle of least privilege by denying access to resources and functionality by default. Explicitly grant access only when necessary and use secure default configurations to minimize exposure. |
| 1. Adhere to the Principle of Least Privilege | Restrict permissions to the minimum necessary for users and processes to function. Regularly audit and update permissions as requirements change to minimize the risk of unauthorized access or privilege escalation. |
| 1. Sanitize Data Sent to Other Systems | Thoroughly sanitize any data being transferred or shared between systems to prevent data leaks and cross-system vulnerabilities. Use data encoding and escaping techniques to prevent injection attacks, particularly when interacting with databases or external APIs. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls to reduce the impact of potential breaches. Combine technical, administrative, and physical controls to create overlapping defenses that protect against various attack vectors. |
| 1. Use Effective Quality Assurance Techniques | Perform rigorous testing, including unit testing, integration testing, static code analysis, and vulnerability scanning. Ensure that test cases cover edge cases and potential security vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards, such as the SEI CERT C++ Coding Standard, to enforce consistency and security practices across the codebase. Periodically review and update standards to align with emerging threats and industry best practices. |

### 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] | This standard ensures that integer operations do not result in overflow, underflow, or truncation errors. Such errors can lead to vulnerabilities, data corruption, or unexpected behavior, compromising software reliability and security. The compliant code should carefully check value ranges and implement type-safe conversions to avoid data loss or unpredictable results. |

| **Noncompliant Code** |
| --- |
| This code attempts to validate an integer against enumeration values but fails to ensure the integer is within the valid range before the type cast, potentially leading to incorrect or unexpected enumeration values. |
| int f(int value) {  // Checks whether a given value is within range of  // acceptable enumeration values. After casting the type  // it might not be able to represent the given integer  // value.  if ((value >= 0) && (value <= 6)) {  return static\_cast<int>(static\_cast<day>(value));  }  return -1;  } |

| **Compliant Code** |
| --- |
| This code correctly validates the integer against the enumeration's valid range before casting, guaranteeing that the resulting enumeration value is valid and defined. |
| int f(int value) {  // The compliant solution checks the value represented  // by the enumeration type before performing the  // conversion to guarantee the conversion doesn’t result  // in an unspecified value. In turn is restricts the  // converted value to one specific enumerator type.  if ((value >= static\_cast<int>(day::Monday)) &&  (value <= static\_cast<int>(day::Sunday))) {  return static\_cast<int>(static\_cast<day>(value));  }  return -1;  } |

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

| **Principles(s):** Validate Input Data – To ensure conversions have proper input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.17.0 | CERT, MISRA | Static analysis tool focused on detecting bugs and enforcing coding standards like CERT and MISRA. |
| Parasoft C/C++test | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | This standard ensures that iterator operations are performed safely, particularly when modifying containers. Failure to update iterators properly after insertion or deletion can lead to undefined behavior or crashes. |

| **Noncompliant Code** |
| --- |
| This code correctly validates the integer against the enumeration's valid range before casting, guaranteeing that the resulting enumeration value is valid and defined. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> vec = {1, 2, 3, 4, 5};  auto pos = vec.begin();  for (int i = 0; i < vec.size(); ++i) {  if (vec[i] == 3) {  pos = vec.insert(pos, 99);  }  std::cout << \*pos++ << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This code avoids iterator invalidation by ensuring pos is assigned a valid iterator after each insertion, thus preventing undefined behavior during iteration. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> vec = {1, 2, 3, 4, 5};  auto pos = vec.begin();  for (int i = 0; i < vec.size(); ++i) {  if (vec[i] == 3) {  pos = vec.insert(pos, 99);  pos++;  } else {  pos++;  }  }  for (int val : vec) {  std::cout << val << std::endl;  }  return 0;  } |

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

| **Principles(s):** Developing Secure Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.17.0 | CERT, MISRA | Static analysis tool focused on detecting bugs and enforcing coding standards like CERT and MISRA. |
| SonarQube/ C++ Plugin | 2.2.0 | Integrated C++ checkers | Integrates C++ static analysis tools and rule sets into the SonarQube dashboard. |
| Parasoft C/C++test | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensure sufficient space allocation for strings and the null terminator to prevent buffer overflow vulnerabilities. Utilize standard library functions like std::string to manage dynamic string data securely. |

| **Noncompliant Code** |
| --- |
| This code is vulnerable to buffer overflows because it uses a fixed-size character array (buff) to read input without any bounds checking, which can lead to writing beyond the array's allocated memory. |
| #include <iostream>  int main() {  char buff[10];  std::cin >> buff;  std::cout << buff << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This code mitigates buffer overflows by using std::string, which dynamically manages memory allocation, thus safely handling input of any size without the risk of overwriting memory. |
| #include <iostream>  #include <string>  int main() {  std::string input;  std::cin >> input;  std::cout << input << std::endl;  return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Medium | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2025.6 | CWE, MISRA | Identifies defects using advanced static analysis with support for CWE, OWASP, and MISRA standards. |
| Parasoft | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |
| TrustInSoft Analyzer | 2025.1 | Formal verification | Performs exhaustive formal verification using mathematical proofs and abstract interpretation. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP | Avoid injecting raw strings into SQL queries. Use parameterized queries to reduce the risk of SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| This code is susceptible to SQL injection. It constructs a SQL query by directly embedding user-provided input, allowing malicious users to inject arbitrary SQL code and potentially compromise the database. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  int main() {  MYSQL \*conn;  MYSQL\_RES \*res;  MYSQL\_ROW row;  conn = mysql\_init(NULL);  mysql\_real\_connect(conn, "host", "user", "password", "database", 0, NULL, 0);  std::string user\_input;  std::cout << "Enter username: ";  std::getline(std::cin, user\_input);  std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "'";  if (mysql\_query(conn, query.c\_str())) {  std::cerr << "MySQL query error: " << mysql\_error(conn) << std::endl;  return 1;  }  res = mysql\_store\_result(conn);  while ((row = mysql\_fetch\_row(res))) {  std::cout << "Username: " << row[0] << ", Password: " << row[1] << std::endl;  }  mysql\_free\_result(res);  mysql\_close(conn);  return 0;  } |

| **Compliant Code** |
| --- |
| This code prevents SQL injection by using parameterized queries with prepared statements. It separates SQL code from user input, treating the input as data rather than executable code, thereby protecting the database from injection attacks. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  int main() {  MYSQL \*conn;  MYSQL\_STMT \*stmt;  MYSQL\_BIND bind[1];  MYSQL\_RES \*res;  MYSQL\_ROW row;  conn = mysql\_init(NULL);  mysql\_real\_connect(conn, "host", "user", "password", "database", 0, NULL, 0);  std::string user\_input;  std::cout << "Enter username: ";  std::getline(std::cin, user\_input);  std::string query = "SELECT \* FROM users WHERE username = ?";  stmt = mysql\_stmt\_init(conn);  if (mysql\_stmt\_prepare(stmt, query.c\_str(), query.length())) {  std::cerr << "MySQL prepare error: " << mysql\_stmt\_error(stmt) << std::endl;  return 1;  }  bind[0].buffer\_type = MYSQL\_TYPE\_STRING;  bind[0].buffer = (char \*)user\_input.c\_str();  bind[0].length = (unsigned long \*)user\_input.length();  bind[0].is\_null = 0;  bind[0].length\_value\_type = MYSQL\_LENGTH\_TYPE\_LONG;  if (mysql\_stmt\_bind\_param(stmt, bind)) {  std::cerr << "MySQL bind param error: " << mysql\_stmt\_error(stmt) << std::endl;  return 1;  }  if (mysql\_stmt\_execute(stmt)) {  std::cerr << "MySQL execute error: " << mysql\_stmt\_error(stmt) << std::endl;  return 1;  }  res = mysql\_stmt\_result\_metadata(stmt);  while ((row = mysql\_fetch\_row(res))) {  std::cout << "Username: " << row[0] << ", Password: " << row[1] << std::endl;  }  mysql\_free\_result(res);  mysql\_stmt\_close(stmt);  mysql\_close(conn);  return 0;  } |

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

| **Principles(s):** Validate Input Data, Adopt Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2025.6 | CWE, MISRA | Identifies defects using advanced static analysis with support for CWE, OWASP, and MISRA standards. |
| Findbugs | 3.0.1 | Bug pattern matching | Analyzes Java bytecode for common bug patterns; not updated since 2015. |
| CppCheck | 2.17.0 | CERT, MISRA | Static analysis tool focused on detecting bugs and enforcing coding standards like CERT and MISRA. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Prevent accessing memory after it has been freed, as it can result in unpredictable behavior or privilege escalation. |

| **Noncompliant Code** |
| --- |
| This code exhibits a use-after-free vulnerability. It dereferences a pointer (ptr) after the memory it points to has been deallocated, which can lead to unpredictable behavior, data corruption, or potential security exploits. |
| #include <iostream>  int main() {  int\* ptr = new int(10);  std::cout << "Value: " << \*ptr << std::endl;  delete ptr;  std::cout << "Value after delete: " << \*ptr << std::endl; // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| This code avoids use-after-free errors by setting the pointer ptr to nullptr immediately after deallocating the memory. It also adds a check to make sure the pointer is not dereferenced after being deleted. |
| #include <iostream>  int main() {  int\* ptr = new int(10);  std::cout << "Value: " << \*ptr << std::endl;  delete ptr;  ptr = nullptr; // Set pointer to null after deleting  if (ptr != nullptr) {  std::cout << "Value after delete: " << \*ptr << std::endl;  }  return 0;  } |

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

| **Principles(s):** Heed Compiler Warnings, Default Deny, Adhere to the Principle of Least Privilege, Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17.0 | CERT, MISRA | Static analysis tool focused on detecting bugs and enforcing coding standards like CERT and MISRA. |
| Parasoft C/C++test | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |
| TrustInSoft Analyzer | 2025.1 | Formal verification | Performs exhaustive formal verification using mathematical proofs and abstract interpretation. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are useful for debugging but should not be present in production code, as they may expose sensitive data or crash applications unexpectedly. |

| **Noncompliant Code** |
| --- |
| This code includes an assertion that could trigger in production. Assertions should only be used during development and debugging, not in production code, as they can cause unexpected program termination or expose sensitive information. |
| #include <cassert>  #include <iostream>  int main() {  int x = 5;  assert(x == 10);  std::cout << "Program continues..." << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This code removes the unnecessary assertion. Production code should handle errors and exceptional conditions explicitly rather than relying on assertions. |
| #include <iostream>  int main() {  int x = 5;  if (x != 10) {  std::cerr << "Error: x is not 10" << std::endl;  return 1;  }  std::cout << "Program continues..." << std::endl;  return 0;  } |

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

| **Principles(s):** Heed Compiler Warnings, Adopt Secure Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 21.0.0git | Clang Static Analyzer | Uses the Clang Static Analyzer for symbolic execution and bug detection. |
| LDRA Tool | 2025.1 | MISRA C:2025 | Provides comprehensive static and dynamic analysis for compliance with MISRA and safety standards. |
| Axivion Suite | 7.9 | CERT, CWE, MISRA | Performs static analysis with rule checks for CWE, CERT, and MISRA; detects architecture violations. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Ensure that exceptions are properly handled and do not cause program termination unexpectedly. Use structured exception handling to catch, log, and manage exceptions, especially at the entry points of the application. |

| **Noncompliant Code** |
| --- |
| This code lacks proper exception handling. Exceptions thrown by throwing\_func() are not caught, which results in program termination due to an unhandled exception. |
| #include <iostream>  #include <stdexcept>  void throwing\_func() {  throw std::runtime\_error("Exception from throwing\_func");  }  void f() {  throwing\_func();  }  int main() {  f();  return 0;  } |

| **Compliant Code** |
| --- |
| This code implements exception handling using a try-catch block. This ensures that exceptions are caught and handled gracefully, preventing program termination and allowing for resource cleanup or error reporting. |
| #include <iostream>  #include <stdexcept>  void throwing\_func() {  throw std::runtime\_error("Exception from throwing\_func");  }  void f() {  throwing\_func();  }  int main() {  try {  f();  } catch (const std::exception& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  return 1;  }  return 0;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2025.1 | MISRA C:2012 Amd 2 | Detects security, safety, and reliability issues with support for MISRA and CWE standards. |
| LDRA Tool | 2025.1 | MISRA C:2025 | Provides comprehensive static and dynamic analysis for compliance with MISRA and safety standards. |
| CodeSonar | 9.0.0 | MISRA, AUTOSAR | Whole-program static analysis with deep semantic analysis and checks for MISRA and AUTOSAR. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Container** | [STD-008-CPP] | Ensure that iterators remain valid after performing container modifications. Invalid iterators can cause undefined behavior, crashes, or data corruption |

| **Noncompliant Code** |
| --- |
| This code uses iterators in an incorrect order with std::for\_each. Using vec.end() as the beginning iterator and vec.begin() as the ending iterator results in undefined behavior because the algorithm expects iterators to define a valid range. |
| #include <iostream>  #include <vector>  #include <algorithm>  int main() {  std::vector<int> vec = {1, 2, 3, 4, 5};  std::for\_each(vec.end(), vec.begin(),(int n){  std::cout << n << std::endl;  });  return 0;  } |

| **Compliant Code** |
| --- |
| This code corrects the iterator order in std::for\_each, using vec.begin() as the beginning iterator and vec.end() as the ending iterator to define a valid range for the algorithm. |
| #include <iostream>  #include <vector>  #include <algorithm>  int main() {  std::vector<int> vec = {1, 2, 3, 4, 5};  std::for\_each(vec.begin(), vec.end(),(int n){  std::cout << n << std::endl;  });  return 0;  } |

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

| **Principles(s):** Keep it Simple |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ Test | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |
| LDRA Tool | 2025.1 | MISRA C:2025 | Provides comprehensive static and dynamic analysis for compliance with MISRA and safety standards. |
| Astree | 23.04 | MISRA, concurrency rules | Uses abstract interpretation to prove absence of runtime errors and enforce MISRA/CERT compliance. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Environment** | [STD-009-CPP] | Avoid using system() calls for executing shell commands, as they pose significant security risks due to command injection vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code uses the system() function to execute shell commands, which poses a significant security risk. It allows for command injection vulnerabilities, where malicious input can be used to execute arbitrary commands on the system. |
| #include <iostream>  #include <string>  int main() {  std::string command;  std::cout << "Enter command: ";  std::getline(std::cin, command);  system(command.c\_str());  return 0;  } |

| **Compliant Code** |
| --- |
| This code avoids the use of system() and uses the Windows API CreateProcessA() instead. This provides a more secure way to create new processes and reduces the risk of command injection. |
| #include <iostream>  #include <string>  #include <Windows.h>  int main() {  std::string command;  std::cout << "Enter application to run: ";  std::getline(std::cin, command);  STARTUPINFOA siStartupInfo;  PROCESS\_INFORMATION piProcessInfo;  memset(&siStartupInfo, 0, sizeof(siStartupInfo));  siStartupInfo.cb = sizeof(siStartupInfo);  memset(&piProcessInfo, 0, sizeof(piProcessInfo));  if (!CreateProcessA(  NULL,  (LPSTR)command.c\_str(),  NULL,  NULL,  FALSE,  CREATE\_NEW\_CONSOLE,  NULL,  NULL,  &siStartupInfo,  &piProcessInfo  )) {  std::cerr << "CreateProcess failed (" << GetLastError() << ")" << std::endl;  return 1;  }  WaitForSingleObject(piProcessInfo.hProcess, INFINITE);  CloseHandle(piProcessInfo.hProcess);  CloseHandle(piProcessInfo.hThread);  return 0;  } |

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

| **Principles(s):** Architect and Design for Security Policies, Keep It Simple, Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |
| PVS-Studio | 7.36 | MISRA C/C++ 2023 | Static analyzer for detecting bugs and vulnerabilities, supports MISRA and SEI CERT rules. |
| Astree | 23.04 | MISRA, concurrency rules | Uses abstract interpretation to prove absence of runtime errors and enforce MISRA/CERT compliance. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Pseudo -randomization** | [STD-010-CPP] | Avoid using pseudo-random number generators like std::rand() for security-sensitive operations. Use cryptographically secure libraries to generate random number |

| **Noncompliant Code** |
| --- |
| This code uses std::rand() to generate a random ID, which is cryptographically insecure. std::rand() produces predictable pseudo-random numbers, making it unsuitable for security-sensitive applications. |
| #include <iostream>  #include <cstdlib>  int main() {  int random\_id = std::rand();  std::cout << "Generated ID: " << random\_id << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This code generates a cryptographically secure random ID using the C++ standard library's random number generation facilities, including std::random\_device, std::mt19937, and std::uniform\_int\_distribution. This approach provides much stronger randomness and avoids predictability. |
| #include <iostream>  #include <random>  int main() {  std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> dist(1, 1000000); // Range for the ID  int random\_id = dist(gen);  std::cout << "Generated secure ID: " << random\_id << std::endl;  return 0;  } |

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

| **Principles(s):** Adopt Secure Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 21.0.0git | Clang Static Analyzer | Uses the Clang Static Analyzer for symbolic execution and bug detection. |
| CodeSonar | 9.0.0 | MISRA, AUTOSAR | Whole-program static analysis with deep semantic analysis and checks for MISRA and AUTOSAR. |
| Parasoft C/C++ Test | 2025.1 | MISRA C:2025 | Performs static and unit testing for C/C++ with full MISRA compliance and coverage metrics. |

### 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.

Existing DevOps policies provide a solid foundation for implementing a DevSecOps platform. While DevOps excels at rapidly delivering software, it often falls short in addressing the growing security concerns of modern organizations. DevSecOps improves upon the DevOps model by integrating security from the very beginning, fostering a culture shift that treats security, development, and operations as equal partners.

The primary goal of DevSecOps is to apply security principles throughout all phases of the Software Development Lifecycle (SDLC), as illustrated in the figure above.

To align with the standards introduced in this policy, I would propose several modifications across each phase of the traditional DevOps lifecycle to elevate it to DevSecOps standards:

* **Planning Phase:** I would incorporate threat modeling by identifying common attack patterns—such as SQL injection or man-in-the-middle attacks—and design countermeasures from the outset.
* **Develop and Build Phases:** Secure coding practices would be enforced, guided by our coding standards and mitigation strategies. This reduces vulnerabilities and limits the potential impact of external threats on the application.
* **Testing Phase:** I would adopt automated unit testing to validate individual components. Additionally, mid-tier integration testing would be implemented to evaluate the full application stack—from the front end to the database—using simulated attacks, including SQL injection and memory-based exploits.
* **Release, Deploy, Operate, and Monitor Phases:** I would utilize secure containerization to prevent unauthorized access to the underlying operating system. Automated log collection and analysis would be employed to detect and mitigate threats in real time. Furthermore, network traffic analysis would be used to identify abnormal spikes targeting specific nodes, enabling early detection of potential DDoS attacks or intrusion attempts.

By embedding security throughout the SDLC, we can create more resilient, secure, and stable applications that are better equipped to handle modern threats.

### 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 | Low | Likely | Medium | Low | 2 |
| STD-002-CPP | Low | Likely | Medium | Low | 2 |
| STD-003-CPP | High | Likely | Medium | Medium | 1 |
| STD-004-CPP | High | Priority | Medium | High | 3 |
| STD-005-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-006-CPP | Low | Unlikely | Low | Low | 1 |
| STD-007-CPP | Low | Priority | Medium | Low | 3 |
| STD-008-CPP | Low | Priority | High | Low | 3 |
| STD-009-CPP | High | Likely | High | Medium | 2 |
| STD-010-CPP | Medium | Likely | Medium | Medium | 3 |

### 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 protection of data that is stored and not actively being used—such as data on a hard drive or in a database. The goal of this policy is to ensure that, in the event of a data breach, stolen files remain inaccessible without the appropriate decryption keys. Attackers would need to either brute-force the encryption, which is time-consuming and computationally expensive, or obtain the encryption keys. This significantly delays or prevents the misuse of the data, reducing the impact of the breach. |
| Encryption in flight | **Encryption-in-flight** involves securing data as it travels across a network. For example, this applies when a web application retrieves information from a database. This phase is particularly vulnerable to interception, making encryption critical. Secure data transmission is typically achieved through SSL/TLS protocols, ensuring encrypted communication between systems such as web servers and databases. Additional layers of protection, like Virtual Private Networks (VPNs), further secure network segments and prevent packet sniffing tools like Wireshark or TCPDump from reading or reconstructing the transmitted data. |
| Encryption in use | **Encryption-in-use** pertains to protecting data while it is actively being processed or accessed. For example, when a web server retrieves data from a database and performs operations on it before presenting it to the user. This form of encryption safeguards data in memory, reducing the risk of exposure during execution. Protection mechanisms may include the use of secure memory handling and programming techniques such as those found in frameworks like .NET’s ProtectedMemory class. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Authentication** is the process by which a system verifies the identity of a user or application. It determines who should and should not have access to a system. This is typically enforced through a logon mechanism tied to credentials such as an email address or username, and verified by a password. Additional methods like two-factor authentication (2FA), one-time passwords (OTP), or OAuth-based single sign-on (SSO) can strengthen this process. The policy ensures that only verified users gain access and that sensitive resources are protected behind authentication gates, minimizing unauthorized access risks. |
| Authorization | **Authorization** follows authentication and determines what level of access an authenticated user has to system resources, such as files or services. It typically uses role-based access controls to enforce permissions. A key principle in this context is **Default Deny**, meaning users should not have access to any resources unless explicitly granted. Authorization is critical in ensuring that authenticated users do not overstep their intended access levels. This policy enforces proper access boundaries and supports secure privilege escalation only through intentional role assignment. |
| Accounting | **Accounting** involves tracking user activity within a system—such as which files are accessed and what changes are made to resources like databases. This audit trail is essential for both operational visibility and forensic analysis following a breach. Tools like Thycotic Secret Server provide centralized logging and reporting for such activities. Having detailed logs allows security teams to detect suspicious behavior, investigate breaches, and conduct root cause analysis quickly. This policy ensures accountability by monitoring system interactions, helping to detect unauthorized access and prevent future incidents. |

**\***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 |  |
| 1.1 | 03/22/2025 | First Milestone Completion | Andy Martinez | Andy Martinez |
| 1.2 | 04/14/2025 | Security Policy Completion | Andy Martinez | Andy Martinez |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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