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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Always check every piece of external input. Do not only make sure its format and expected values are correct, but also check both its structure and its meaning. Use whitelist methods to strictly limit allowed characters, lengths, and types. Do validations on both the client side and the server side, and include strong error handling to deal with unexpected or harmful input. This careful practice lowers the risk of issues like SQL injection, cross-site scripting, and buffer overflow problems. |
| 1. Heed Compiler Warnings | Treat each compiler warning as a sign that there might be hidden security problems. Turn on all compiler warnings when you build the code and use static analysis tools to find and rank warnings that could affect security. Write down any issues that come up repeatedly and set up a way to check and fix these warnings before the code goes live. This helps keep the code clean and lowers the chance of security problems from ignored warnings. |
| 1. Architect and Design for Security Policies | Include security from the start by doing threat modeling and risk analysis early in the project. Use design patterns and system architectures that ensure proper access controls, protect data, and manage errors. Clearly record all security requirements along with system diagrams. This way, security is built into the system from the beginning, reducing the risk of vulnerabilities later on. |
| 1. Keep It Simple | Keep things simple by avoiding complex logic and extra features. Use clear, modular code that is easier to audit, test, and maintain. This simplicity makes the code easier to read and lowers the chance of bugs and security issues. Follow consistent coding practices and limit dependencies so the code stays secure and manageable over time. |
| 1. Default Deny | Use a strict default-deny approach by making sure every access decision is explicitly approved. Don't assume any permissions unless strong authentication and authorization checks have confirmed them. This method lowers exposure by denying all requests unless they meet a pre-approved set of rules. Record every exception with clear reasons and review them regularly to ensure continued compliance. |
| 1. Adhere to the Principle of Least Privilege | Keep all user and process permissions at the lowest levels needed for work. This rule reduces the harm caused by mistakes or attacks and limits any security breaches to as small an area as possible. Use role-based access controls (RBAC) and check access rights regularly to make sure they match current job roles. Also, use automated tools that watch for and alert you about any increases in privileges. |
| 1. Sanitize Data Sent to Other Systems | Beyond just checking inputs, always clean up data before sending it to external systems. Use methods like encoding, escaping, or formatting that fit the target system to stop injection attacks and prevent data leaks. Regularly update these cleaning steps based on new threats and system changes. This routine should be part of a larger strategy for handling data securely, which includes end-to-end encryption and strict access controls. |
| 1. Practice Defense in Depth | Use several independent security controls to protect your critical assets. You can include network firewalls, intrusion detection systems, application firewalls, and endpoint security measures. Each layer acts as a backup if one fails, lowering the overall risk. Write down what each layer is for and how it's set up, and test them regularly—both automatically and manually—to make sure your defense strategy works. |
| 1. Use Effective Quality Assurance Techniques | Include strong testing strategies in every phase of software development. Use static and dynamic analysis tools, unit tests, integration tests, and code reviews to spot security gaps. Make quality assurance an ongoing step by integrating testing into your CI/CD pipeline so you can catch vulnerabilities early. Also, focus on peer reviews and automated alerts to help maintain security at each stage of development. |
| 1. Adopt a Secure Coding Standard | Adopt a secure coding standard for the entire organization, and update it regularly to cover the newest threats and best practices. Provide training and documentation so every developer understands these rules. Use automated checks in your build process to flag any code that doesn't follow the standards before it goes into production. This approach builds a security-focused culture, ensuring that safe coding practices are an essential part of every development step. |

### 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** | **Strong Type Checking** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Enforcing strong type checking can stop many programming errors that might lead to security vulnerabilities, such as buffer overflows and type confusions. |

| **Noncompliant Code** |
| --- |
| Using integers to handle characters can cause type mismatch problems. |
| int c = 'A'; *// Improper use of integer for character.* |

| **Compliant Code** |
| --- |
| Use char type for characters to ensure type safety. |
| char c = 'A'; *// Correct type for characters.* |

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

| **Principles(s):**   * **Adopt a Secure Coding Standard (Principle 10):** Using proper types supports secure coding practices. Heed * **Compiler Warnings (Principle 2):** Strong type checking mitigates issues that otherwise might be dismissed as warnings. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | C++ Type Safety | Identifies unsafe type conversions and implicit type coercions. |
| Cppcheck | 2.9 | strongTypeChecks | Verifies correct type usage and flags usage of incompatible types. |
| Coverity | 2024.1 | Type Checking Analyzer | Scans for type mismatches and improper conversions in variable assignments. |
| Helix QAC | 2024.4 | Type Consistency Checker | Makes sure that type declarations are consistent with secure coding practices. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate External Values** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Checking all external inputs and data values helps prevent attacks like SQL injection and cross-site scripting. |

| **Noncompliant Code** |
| --- |
| Assuming user input is safe without validation. |
| std::string input;  std::cin >> input; *// Risk of SQL injection if used in a query without validation.* |

| **Compliant Code** |
| --- |
| Check that the input matches the expected safe patterns. |
| std::string input;  std::cin >> input;  validateInput(input); *// Assume validateInput() checks for safe content.* |

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

| **Principles(s):**   * **Validate Input Data (Principle 1): Ensures** that external data conforms to expected formats and values. * **Sanitize Data Sent to Other Systems (Principle 7):** Reduces risks when using user-provided input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | External Input Validation | Flags usage of unvalidated external input that might lead to injection attacks. |
| Coverity | 2024.1 | InputSanitizer | Checks data flow from external inputs to ensure that proper validations occur. |
| Cppcheck | 2.9 | inputValidation | Identifies patterns where external inputs are used without checks. |
| Fortify Static | 20.2 | SQL Injection Checker | Scans the code for vulnerabilities related to direct concatenation in queries. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Safe String Handling** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Handle strings properly to avoid buffer overflows and protect data integrity. |

| **Noncompliant Code** |
| --- |
| Using standard C string functions that do not perform boundary checks. |
| char buffer[10];  strcpy(buffer, "This is a long string that overflows the buffer."); |

| **Compliant Code** |
| --- |
| Using safer string functions that include bounds checking. |
| char buffer[10];  strncpy(buffer, "Safe string", sizeof(buffer));  buffer[sizeof(buffer) - 1] = '\0'; *// Ensure null termination.* |

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

| **Principles(s):**   * **Keep It Simple (Principle 4):** Using safe string functions simplifies code and reduces risk. Adopt a Secure * **Coding Standard (Principle 10):** Encourages the use of bounds-checking functions to prevent overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | bufferOverflow | Detects potential buffer overflows by checking bounds on string functions. |
| SonarQube | 9.9 | Safe String Handling | Flags unsafe string operations and suggests secure alternatives. |
| Coverity | 2024.1 | String Handling Analyzer | Identifies risky string manipulation patterns that can lead to memory safety issues. |
| Helix QAC | 2024.4 | Buffer Safety Checker | Scans for out-of-bound read/write issues in string operations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL injection by cleaning inputs and using parameterized queries. |

| **Noncompliant Code** |
| --- |
| Create SQL queries directly from user input. |
| std::string user\_input;  std::string query = "SELECT \* FROM users WHERE name = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| Using parameterized queries to prevent SQL injection. |
| std::string user\_input;  sql::PreparedStatement \*pstmt = con->prepareStatement("SELECT \* FROM users WHERE name = ?");  pstmt->setString(1, user\_input);  pstmt->executeQuery(); |

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

| **Principles(s):**   * **Validate Input Data (Principle 1):** Always ensure input is safe before using it in queries. Sanitize Data Sent to * **Other Systems (Principle 7):** Prepares inputs in a controlled manner to avoid injection vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | SQL Injection Prevention | Flags concatenated SQL queries and recommends the use of parameterized queries. |
| Fortify Static | 20.2 | SQL Injection Checker | Detects risky string concatenation patterns leading to SQL injection vulnerabilities. |
| Coverity | 2024.1 | SQL Vulnerability Analyzer | Analyzes code for direct usage of unsafe SQL query methods. |
| Helix QAC | 2024.2 | Parameterized Query Enforcement | Makes sure that queries are built using secure, parameterized interfaces. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Secure Memory Management** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Managing memory securely to avoid leaks and unauthorized access. |

| **Noncompliant Code** |
| --- |
| Improper memory allocation and deallocation can lead to memory leaks. |
| int \*ptr = new int[10]; *// Forgot to delete allocated memory* |

| **Compliant Code** |
| --- |
| Making sure all allocated memory is properly freed. |
| int \*ptr = new int[10];  delete [] ptr; *// Correct deallocation of memory.* |

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

| **Principles(s):**   * **Practice Defense in Depth (Principle 8):** Proper memory management helps prevent additional vulnerabilities. * **Heed Compiler Warnings (Principle 2):** Reduces risks signaled by static analysis regarding memory usage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18 | Memory Leak Detector | Detects unfreed dynamic memory leading to leaks. |
| Coverity | 2024.1 | Memory Management Check | Identifies mismatched allocation/deallocation patterns. |
| Cppcheck | 2.9 | MemoryLeak | Flags potential memory leaks in resource allocation. |
| Helix QAC | 2024.2 | Resource Cleanup Analyzer | Makes sure proper deallocation and resource management practices. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Proper Use of Assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use assertions to detect conditions that should never happen in production code, as a way to find critical errors during development. |

| **Noncompliant Code** |
| --- |
| Misusing assertions to handle runtime errors. |
| assert(user != nullptr); *// Misuse as it may be disabled in production* |

| **Compliant Code** |
| --- |
| Assertions should only be used to check for programmer errors. |
| assert(ptr != nullptr && "Pointer must not be null"); *// Proper use of assert for debugging.* |

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

| **Principles(s):**   * **Adopt a Secure Coding Standard (Principle 10):** Asserts are for detecting programmer errors during development, not a replacement for runtime error handling. * **Heed Compiler Warnings (Principle 2):** Early detection via assertions can catch issues before they reach production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | Assertion Misuse Checker | Detects reliance on assertions in places that should have proper error handling. |
| SonarQube | 9.9 | Assertion Usage | Flags inappropriate use of assertions in production code. |
| Coverity | 2024.1 | Assertion Analyzer | Examines assertion patterns and recommends improvements for safer error detection. |
| Helix QAC | 2024.4 | Debug Assertion Checker | Verifies correct implementation and usage context of assertions. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Robust Exception Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Make sure that exceptions are used to handle errors in a controlled and predictable manner. |

| **Noncompliant Code** |
| --- |
| Allowing exceptions to propagate without handling. |
| int divide(int a, int b) {  return a / b; *// Throws divide by zero exception without handling*  } |

| **Compliant Code** |
| --- |
| Handling exceptions to maintain application stability. |
| int divide(int a, int b) {  if (b == 0) throw std::invalid\_argument("divider cannot be zero");  return a / b;  } |

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

| **Principles(s):**   * **Architect and Design for Security Policies (Principle 3):** Robust exception handling helps catch errors early and prevents cascading failures. * **Practice Defense in Depth (Principle 8**): Centralized error management is one layer among multiple safeguards. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.1 | Exception Handling Analysis | Detects unhandled exceptions and recommends appropriate error management. |
| SonarQube | 9.9 | Exception Handling Best Practices | Identifies patterns where exceptions are not properly caught or managed. |
| Cppcheck | 2.9 | ExceptionHandlingCheck | Checks code paths for proper exception catch blocks and error handling sequences. |
| Helix QAC | 2024.4 | Runtime Exception Verifier | Flags missing try-catch blocks and improper propagation of exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Resource Acquisition Is Initialization (RAII)** |
| --- | --- | --- |
| Memory Management | STD-008-CPP | RAII is a C++ programming idiom that makes sure resource acquisition is tied to the lifetime of an object. It is for handling resource management, including memory, network, and hardware connections, which prevents resource leaks and guarantees that cleanup logic is always called. |

| **Noncompliant Code** |
| --- |
| Manually managing resource without RAII can lead to leaks when exceptions are thrown and not caught correctly. |
| FILE\* file = fopen("example.txt", "r");  if (error\_occurs()) {  return; // File is never closed if an error occurs  }  fclose(file); |

| **Compliant Code** |
| --- |
| Using RAII makes sure that resources are correctly released in accordance with an object lifetime. |
| std::ifstream file("example.txt"); *// File is automatically closed when file object goes out of scope* |

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

| **Principles(s):**   * **Keep It Simple (Principle 4):** RAII simplifies resource management. * **Practice Defense in Depth (Principle 8):** It adds a layer of protection by ensuring cleanup happens regardless of exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | RAII Compliance Checker | Detects instances where RAII patterns are not used for resource management. |
| SonarQube | 9.9 | RAII Pattern Usage | Flags manual resource management that should be replaced with RAII idioms. |
| Coverity | 2024.1 | Resource Management Analyzer | Analyzes code for proper use of RAII for various resource types. |
| Helix QAC | 2024.2 | RAII Enforcement Checker | Makes sure that objects managing resources correctly release them upon destruction. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use Exceptions for Error Handling** |
| --- | --- | --- |
| Error Handling | STD-009-CPP | Using exceptions for error handling in C++ makes sure that errors are handled at a known point in the program, improving the maintainability and reliability of code. |

| **Noncompliant Code** |
| --- |
| Using error codes for error handling which can be ignored or mishandled by developers. |
| int status = processData();  if (status != 0) {  *// Error handling might be incomplete or ignored*  } |

| **Compliant Code** |
| --- |
| Adding exception handling offers a clear way to manage errors that can't be overlooked. |
| try {  processData();  } catch (const std::runtime\_error& e) {  *// Handle error*  } |

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

| **Principles(s):**   * **Architect and Design for Security Policies (Principle 3):** Centralizes error handling and improves overall stability. * **Use Effective Quality Assurance Techniques (Principle 9):** Promotes maintainability and clarity in error management practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.1 | Exception Handling Analyzer | Reviews code for proper and consistent exception handling throughout the codebase. |
| SonarQube | 9.9 | Exception Management | Detects error-handling patterns that may result in unhandled exceptions. |
| Cppcheck | 2.9 | exceptionHandling | Verifies that exceptions are correctly caught and handled. |
| Helix QAC | 2024.4 | Exception Flow Checker | Analyzes the propagation of exceptions and flags potential risk areas. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use of Lock Hierarchies** |
| --- | --- | --- |
| Concurrency | STD-010-CPP | Lock hierarchies help prevent deadlocks in multi-threaded applications by setting a specific order for acquiring and releasing locks. By consistently following this order, you can systematically avoid deadlocks. |

| **Noncompliant Code** |
| --- |
| 12 |
| std::mutex mutexA;  std::mutex mutexB;  void thread1() {  std::lock\_guard<std::mutex> lockA(mutexA);  std::lock\_guard<std::mutex> lockB(mutexB); // Potential for deadlock with thread2  }  void thread2() {  std::lock\_guard<std::mutex> lockB(mutexB);  std::lock\_guard<std::mutex> lockA(mutexA); *// Potential for deadlock with thread1*  } |

| **Compliant Code** |
| --- |
| Consistently ordering lock acquisition prevents potential deadlocks. |
| std::mutex mutexA;  std::mutex mutexB;  void consistentLockOrder() {  std::lock(mutexA, mutexB); *// Lock both mutexes without deadlock*  std::lock\_guard<std::mutex> lockA(mutexA, std::adopt\_lock);  std::lock\_guard<std::mutex> lockB(mutexB, std::adopt\_lock);  *// Perform actions with both resources locked*  } |

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

| **Principles(s):**   * **Default Deny (Principle 5):** A strict lock acquisition order prevents unpredictable access. * **Architect and Design for Security Policies (Principle 3):** Structure in multi-threading minimizes concurrency risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.4 | Concurrency Lock Order Checker | Detects inconsistent lock ordering that may cause deadlocks. |
| Coverity | 2024.1 | Deadlock Detection | Analyzes code for potential deadlock scenarios in multithreaded applications. |
| SonarQube | 9.9 | Concurrency Best Practices | Flags unsafe lock acquisition orders in concurrent code. |
| Cppcheck | 2.9 | DeadlockPotential | Identifies patterns that can lead to deadlock conditions and recommends consistent lock order. |

### 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.

**Automation in the DevSecOps Lifecycle:**

1. **Pre-production Phases (Assess & Plan, Design, Build, Verify & Test):** 
   1. **Assess & Plan:** Automated tools are introduced early to perform threat modeling and landscape analysis. This phase uses tools to scan for potential security vulnerabilities in design and architecture, ensuring that new features are aligned with the security standards from the start.
   2. **Design:** As code prototypes and system designs are developed, automation—through plugins and integrated checkers—is used to enforce secure design principles and adhere to best practices. This prevents insecure design choices before they reach production.
   3. **Build:** During code compilation and integration, automated static analysis tools scan the code against the defined coding standards. These checks stop the build process if violations are found, ensuring that only compliant code moves forward.
   4. **Verify & Test:** Automated vulnerability scanning (both SAST and DAST) and compliance tests continuously assess whether the evolving code meets internal policies and external regulatory mandates. This stage provides a safeguard that all components comply with coding and system-level such as encryption and access control standards.
2. **Production Phases (Transition, Monitor & Detect, Respond, Maintain & Stabilize):**
   1. **Transition:** Before full deployment, automation verifies that infrastructure-as-code configurations comply with security policies such as proper server setup and network configurations. Automated deployment pipelines ensure that only systems passing all compliance tests are released to production.
   2. **Monitor & Detect:** In production, security information and event management (SIEM) systems continuously ingest logs and alerts. Automated tools monitor for any deviations or suspicious activities, ensuring ongoing compliance with the established policy frameworks.
   3. **Respond:** When a non-compliant event or potential breach is detected, automated incident response systems are in place. These systems can isolate the affected part of the system or block suspicious traffic, ensuring quick remediation while preserving system integrity.
   4. **Maintain & Stabilize:** Continuous automated assessments compare current system states with the security baseline established during earlier phases. If any discrepancies—like configuration drift or unauthorized changes—are identified, automated alerts trigger corrective actions or notify the security team.

**Guidance on Modifying the Existing DevOps Process:**

* **Integrate Automated Checks at Every Stage:** Modify the current DevOps process to incorporate automated security checks during planning, design, build, and testing stages. Ensure that the CI/CD pipeline is extended to include security compliance gates that validate adherence to coding and system standards before allowing code progression.
* **Enhance Existing Infrastructure:** Use the current well-established DevOps infrastructure as the base, then layer in additional automation tools such as static analysis, dynamic scanning, and configuration validation to enforce the defined security policies without disrupting existing workflows.
* **Continuous Monitoring and Real-Time Enforcement:** Adjust monitoring systems to include automated alerting and incident response based on predefined security baselines and threat signatures. This ensures that any deviation from standards is immediately addressed, maintaining continuous compliance.
* **Feedback and Iteration:** Integrate feedback loops within the pipeline so that automated assessments guide developers in real time, fostering rapid iterations that align with both development goals and security requirements.

### 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 | 3 |
| STD-003-CPP | High | Medium | Medium | High | 3 |
| STD-004-CPP | High | Medium | Medium | High | 3 |
| STD-005-CPP | Medium | Medium | Low | Medium | 3 |
| STD-006-CPP | Medium | Low | Low | Medium | 2 |
| STD-007-CPP | High | Medium | Medium | High | 3 |
| STD-008-CPP | High | Low | Low | High | 2 |
| STD-009-CPP | High | Low | Medium | High | 3 |
| STD-010-CPP | Medium | Medium | High | High | 3 |
| SYS-001-ENC | High | Low | Low | High | 2 |
| SYS-002-ENC | High | Medium | Medium | High | 3 |
| SYS-003-ENC | Medium | Low | High | High | 2 |
| SYS-004-AAA | High | Low | Medium | High | 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 means storing data in a protected format so that, if the storage media is lost or stolen, the data remains unreadable to unauthorized parties.  **How to Apply:** All sensitive files, databases, or backups must be encrypted on disk. Keys should be securely managed, rotated regularly, and stored separately from the data.  **Reason to Use:** This policy protects confidentiality and ensures compliance with data privacy regulations. |
| Encryption in flight | Encryption in flight uses secure protocols (like TLS 1.2 or higher) to protect data from eavesdropping or tampering during transmission.  **How to Apply:** All sensitive data transfers—internal or external—must use HTTPS, VPN, or another validated secure channel with valid certificates.  **Reason to Use:** This policy prevents man-in-the-middle attacks and protects data integrity over untrusted networks. |
| Encryption in use | Encryption in use safeguards data that is actively being processed in memory, often via secure enclaves or hardware-assisted encryption.  **How to Apply:** High-risk operations such as handling cryptographic keys or sensitive personal data must minimize plaintext in memory and use trusted execution environments.  **Reason to Use:** This policy reduces the risk of exposing sensitive data to memory dumps or unauthorized runtime access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication confirms a user’s identity, typically via passwords, tokens, or multi-factor methods.  **How to Apply:** All system logins must use strong credentials like 12+ characters and complexity rules as well as MFA where feasible. User identities and login events must be logged for auditing.  **Reason to Use:** This policy makes sure only legitimate, verified users gain access and helps detect unauthorized logins. |
| Authorization | Authorization grants or restricts what authenticated users can do, following the principle of least privilege.  **How to Apply:** Implement role-based access controls (RBAC). Assign each user the minimal set of permissions needed for their job; regularly review and remove unused privileges.  **Reason to Use:** This policy limits damage in case of an account compromise and helps maintain tight control over resource access. |
| Accounting | Accounting involves logging and auditing user activities (logins, file access, database changes) to maintain a traceable record of all critical actions.  **How to Apply:** Store logs in a centralized system for a minimum retention period usually one year. Monitor logs for anomalies and review audit trails regularly.  **Reason to Use:** This policy supports incident investigations, compliance checks, and ensures accountability for all system actions. |

**\***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 | 04/07/2025 | Revised with updated coding standards and risk assessments, improved the Ten Core Security Principles and added automation details | Anthony DePoalo | [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 |