**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 | Validating input data is critical to ensuring that any information received from users, external systems, or untrusted sources is both expected and safe to process. By rigorously checking input formats, lengths, and types, and by filtering out any potentially malicious content, developers can prevent injection attacks, buffer overflows, and other vulnerabilities that stem from improper input handling. |
| 1. Heed Compiler Warnings | Compiler warnings often highlight potential issues or risky constructs that might not cause immediate errors but could lead to security vulnerabilities or unstable behavior later. Paying close attention to these warnings allows developers to address subtle bugs and unsafe practices early in the development process, ultimately leading to more robust and secure code. |
| 1. Architect and Design for Security Policies | Incorporating security policies at the design stage ensures that applications are built with a strong security foundation from the outset. This involves outlining clear rules and protocols for access control, data handling, and threat mitigation. By architecting systems with security in mind, organizations can reduce vulnerabilities, streamline compliance, and make it easier to adapt to evolving threats. |
| 1. Keep It Simple | A simpler codebase and system architecture reduce the likelihood of hidden vulnerabilities and make maintenance easier. Complexity often introduces unintended interactions and obscure bugs, which can be exploited by attackers. By striving for simplicity, developers can implement security measures more effectively and ensure that the system remains understandable and manageable over time. |
| 1. Default Deny | The default deny principle means that, unless explicitly allowed, all access requests should be refused. This conservative approach minimizes the attack surface by ensuring that only authenticated and authorized interactions are permitted. By defaulting to denial, developers force a proactive configuration of permissions and access rules, significantly reducing the risk of unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege dictates that every component of a system, whether it’s a user, process, or service, should have only the minimum privileges necessary to perform its function. By limiting permissions, organizations can contain potential breaches, ensuring that even if an account or process is compromised, the impact is minimized. |
| 1. Sanitize Data Sent to Other Systems | When transferring data between systems, it is crucial to sanitize the information to remove any potentially dangerous content. This practice helps to prevent injection attacks and ensures that data is formatted correctly for the receiving system. By carefully cleaning and validating outbound data, developers can prevent the propagation of vulnerabilities across interconnected systems. |
| 1. Practice Defense in Depth | Defense in depth is a security strategy that layers multiple defensive measures throughout a system. If one layer fails, additional layers continue to provide protection. This comprehensive approach reduces the probability of a successful attack by ensuring that attackers must bypass several, potentially redundant, security controls before compromising critical assets. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance (QA) techniques, including code reviews, automated testing, and static analysis, are essential to identify and rectify security flaws early in the development process. By rigorously testing for vulnerabilities and ensuring adherence to security requirements, organizations can improve software reliability and reduce the likelihood of exploits in production environments. |
| 1. Adopt a Secure Coding Standard | A secure coding standard provides a set of best practices and guidelines that developers can follow to write code that minimizes vulnerabilities. By adhering to a consistent, industry-recognized standard, such as the SEI CERT C++ Coding Standard, teams ensure that security is integrated into every phase of development, leading to a more secure and maintainable codebase. |

### 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] | Ensures that variables and data structures use the correct and explicit data types to prevent implicit conversions, precision loss, and potential vulnerabilities related to type errors. |

| **Noncompliant Code** |
| --- |
| A variable is declared using a generic type with implicit conversion, which might lead to truncation or misinterpretation of data. |
| float value = 3.14159;  int truncated = value; // Implicit conversion causing loss of precision |

| **Compliant Code** |
| --- |
| The variable uses an explicit cast or a type-safe alternative to avoid precision loss and clearly indicate the conversion, making the developer’s intent explicit. |
| float value = 3.14159;  int truncated = static\_cast<int>(value);  // Explicit conversion clarifying the intentional truncation |

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

| **Principles(s):**   * (1) Ensures input data conforms to expected types. * (2) Helps detect implicit type conversions that might cause vulnerabilities. * (9) Identifies data type mismatches and prevents logic errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | v2.12 | TypeMismatch | Detects implicit conversions and type mismatches |
| Clang Static Analyzer | LLVM 17.0.6 | CheckTypeSafety | Identifies unintended type promotions and incorrect type usage |
| SonarQube | V10.3 LTS | S3655 | Finds type inconsistencies in C++ codebases |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Guarantees that data values are validated and within expected ranges before use, reducing the risk of processing invalid or malicious data. |

| **Noncompliant Code** |
| --- |
| Data is accepted from user input without validation, which might allow values outside the expected range, potentially leading to undefined behavior. |
| int getUserInput() {  int input;  std::cin >> input; // No range check<br> return input;  } |

| **Compliant Code** |
| --- |
| The code validates the user input, ensuring the value falls within an acceptable range before using it further in the program. |
| [Compliant code block; code should be indented using 12-point Courier New font.] |

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

| **Principles(s):**   * (1) Ensures only valid data is processed. * (7) Prevents malformed data from propagating to other systems. * (8) Provides multiple layers of validation and checks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | V7.30.72048 | V1001 | Detects improper value assignments and potential integer overflows |
| Clang Static Analyzer | LLVM 17.0.6 | Security Checks | Checks for invalid assumptions in variable value ranges |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensures string operations (such as concatenation, copying, or comparison) are performed safely, reducing risks such as buffer overflows or encoding errors. |

| **Noncompliant Code** |
| --- |
| Direct manipulation of character arrays without proper bounds checking, which may lead to buffer overflows. |
| char buffer[10];  strcpy(buffer, "This is a very long string that exceeds buffer size"); |

| **Compliant Code** |
| --- |
| Using safe string handling functions (or C++ string objects) ensures bounds checking and prevents overflow errors. |
| #include <string>  #include <iostream>  std::string buffer = "This is a safe string assignment"; |

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

| **Principles(s):**   * (1) Prevents unsafe string operations that could lead to buffer overflows. * (7) Ensures proper encoding to prevent unintended behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | LLVM 17.0.6 | CStringChecks | Detects unsafe string manipulations and out-of-bounds accesses |
| Cppcheck | V2.12 | BufferOverrun | Flags dangerous string concatenation and copying patterns |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevents SQL injection attacks by using parameterized queries and prepared statements, ensuring that user input is never concatenated directly into SQL commands. |

| **Noncompliant Code** |
| --- |
| Building SQL query strings by concatenating user input directly into the query, making the application vulnerable to SQL injection. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "';"; |

| **Compliant Code** |
| --- |
| Using parameterized queries or prepared statements avoids direct injection of user input into the SQL command. |
| #include <cppconn/prepared\_statement.h>  // Assuming conn is a valid database connection  std::unique\_ptr<sql::PreparedStatement> pstmt(conn->prepareStatement("SELECT \* FROM users WHERE name = ?"));  pstmt->setString(1, userInput); |

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

| **Principles(s):**   * (1) Ensures input is correctly processed and prevents malicious input. * (7) Prevents injection attacks by filtering and structuring user input. * (8) Uses multiple layers (e.g., parameterized queries, least privilege) to mitigate risks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Highly Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v10.3 LTS | S3649 | Identifies unsafe SQL query construction |
| Fortify Static Code Analyzer | v23.2.0 | SQL\_INJECTION | Detects direct user input concatenation into SQL queries |
| CodeQL | V2.15.1 | cpp/sql-injection | Performs deep analysis of SQL-related vulnerabilities |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Ensures that memory is allocated, accessed, and deallocated correctly to prevent issues like buffer overflows, memory leaks, and use-after-free errors. |

| **Noncompliant Code** |
| --- |
| Dynamically allocated memory is not properly checked for allocation success and is freed without nullifying the pointer, risking dangling pointers. |
| int\* p = new int[10];  // ... use p  delete[] p;  // p is now dangling |

| **Compliant Code** |
| --- |
| Memory allocation is checked, and after deletion, the pointer is set to nullptr to prevent accidental use of freed memory. |
| int\* p = new(std::nothrow) int[10];  if (!p) { throw std::bad\_alloc(); }  // ... use p  delete[] p;  p = nullptr; |

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

| **Principles(s):**   * (4) Reducing complexity makes memory handling safer. * (8) Applying multiple techniques to ensure memory integrity. * (9) Detects memory-related issues before deployment. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | LLVM 17.0.6 | HeapUseAfterFree | Detects memory leaks and use-after-free errors |
| Valgrind | v3.22.0 | Memcheck | Identifies memory access violations and leaks |
| Coverity Static Analysis | v2024.2 | MEMORY\_LEAK | Flags improper memory allocation and deallocation |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Utilizes assertions to verify program invariants during development, ensuring that critical assumptions are valid. However, assertions should not be used for production error handling. |

| **Noncompliant Code** |
| --- |
| Over-relying on assertions for error handling in production code, which may be disabled and thus not enforce critical safety checks. |
| assert(pointer != nullptr); // Assumes pointer is valid even in production |

| **Compliant Code** |
| --- |
| Use assertions for debugging and invariants, but also include proper runtime error handling in production code to safely handle unexpected conditions. |
| assert(pointer != nullptr); // Debugging  if (pointer == nullptr) {  throw std::runtime\_error("Null pointer encountered");  } |

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

| **Principles(s):**   * (9) Identifies logic errors early in development. * (3) Helps enforce expected behaviors at runtime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | V7.30.72048 | V2003 | Detects excessive or improper assertion use |
| Cppcheck | v2.12 | RedundantAssertion | Flags redundant assertions that do not improve security |
| Clang Static Analyzer | LLVM 17.0.6 | AssertionChecker | Identifies assertions that fail due to incorrect assumption |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Mandates consistent use of exception handling to manage error conditions. This helps in creating robust code that can recover gracefully from unexpected runtime issues. |

| **Noncompliant Code** |
| --- |
| Using error codes and manual checks everywhere rather than leveraging structured exception handling, which complicates error propagation and recovery. |
| int result = doSomething();  if (result != 0) {  // handle error manually  } |

| **Compliant Code** |
| --- |
| Employing exceptions to propagate error conditions and handling them in catch blocks provides a clearer separation of normal and error-handling code. |
| try {  doSomething();  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  } |

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

| **Principles(s):**   * (4) Ensures clear and predictable error handling. * (9) Prevents crashes and ensures graceful error recovery. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | LLVM 17.0.6 | ExceptionSafety | Detects unhandled exceptions |
| SonarQube | V10.3 LTS | S3824 | Identifies poor exception handling practices |
| PVS-Studio | V7.30.72048 | V1004 | Flags missing or inconsistent exception management |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety Coding | [STD-008-CPP] | Ensures pointers are initialized, checked, and managed properly to avoid common pitfalls such as dangling pointers, memory leaks, and invalid memory access. |

| **Noncompliant Code** |
| --- |
| Directly dereferencing a raw pointer without ensuring it is not null, which may lead to a segmentation fault if the pointer is invalid. |
| int\* p = nullptr;  // ...  int value = \*p; // Dereferencing without check |

| **Compliant Code** |
| --- |
| Always check that the pointer is not null before dereferencing it, or use smart pointers to manage lifetime automatically. |
| #include <memory>  std::unique\_ptr<int> p = std::make\_unique<int>(10);  if (p) {  int value = \*p;  } |

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

| **Principles(s):**   * (2) Helps detect pointer-related issues early. * (9) Prevents memory corruption due to pointer misuse. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | LLVM 17.0.6 | DanglingPointer | Detects invalid memory accesses and leaks |
| Valgrind | v3.22.0 | InvalidRead | Flags use of uninitialized or dangling pointers |
| Coverity Static Analysis | v2024.2 | NULL\_DEREFERENCE | Identifies unsafe pointer dereferences |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer Overflow Prevention | [STD-009-CPP] | Focuses on validating buffer boundaries and using secure functions to prevent buffer overflow vulnerabilities, a common source of security issues. |

| **Noncompliant Code** |
| --- |
| Using unsafe functions like gets or unchecked strcpy that do not limit the number of characters copied, making the code vulnerable to buffer overflows. |
| char buffer[10];  strcpy(buffer, userInput); |

| **Compliant Code** |
| --- |
| Replace unsafe functions with safer alternatives that enforce buffer size limits, such as fgets or strncpy (or better yet, C++ string classes). |
| char buffer[10];  strncpy(buffer, userInput, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; |

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

| **Principles(s):**   * (1) Ensures buffer sizes are not exceeded. * (8) Uses multiple layers of security to prevent buffer overflows. * (9) Detects and mitigates vulnerabilities before release. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Highly Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | LLVM 17.0.6 | BufferOverflow | Detects out-of-bounds memory access |
| Fortify Static Code Analyzer | V23.2.0 | BUFFER\_OVERFLOW | Identifies unsafe buffer operations |
| AddressSanitizer | LLVM 17.0.6 | StackOverflowCheck | Catches runtime buffer overflows |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Cleanup | [STD-010-CPP] | Ensures that all allocated resources (memory, file handles, sockets, etc.) are properly released even in the case of exceptions or early returns. This prevents resource leaks and other related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Allocating resources without ensuring that they are released if an error occurs, leading to resource leaks. |
| FILE\* file = fopen("data.txt", "r");  if (file == nullptr) { return; }  // Process file, but no fclose on error path |

| **Compliant Code** |
| --- |
| Use RAII (Resource Acquisition Is Initialization) techniques or ensure that cleanup is performed in all code paths, including error handling, to release resources. |
| #include <cstdio>  #include <memory>  struct FileDeleter {  void operator()(FILE\* file) const { if (file) fclose(file); }  };  std::unique\_ptr<FILE, FileDeleter> file(fopen("data.txt", "r"));  if (!file) { throw std::runtime\_error("Failed to open file"); } |

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

| **Principles(s):**   * (6) Ensures resources are properly managed to prevent misuse. * (9) Helps identify leaks and dangling resources. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | v3.22.0 | LeakCheck | Detects resource leaks (memory, file handles, etc.) |
| Cppcheck | v2.12 | ResourceLeak | Identifies missing cleanup in error paths |
| Coverity Static Analysis | V2024.2 | RESOURCE\_LEAK | Flags improper deallocation of resources |

### 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.

The DevSecOps model follows a continuous development cycle represented by an infinity loop, emphasizing ongoing security measures. The process begins with assessing and planning, where security risks and compliance requirements are identified. It then moves into designing and building, ensuring security is embedded from the start. DevSecOps remains at the center of this loop, maintaining system integrity by continuously monitoring, testing, and enforcing security policies.

Automation tools should be incorporated at key stages to enforce compliance, such as automated static and dynamic code analysis during the build phase to detect vulnerabilities early, continuous integration/continuous deployment (CI/CD) pipelines with built-in security checks to prevent insecure code from reaching production, automated compliance audits to track adherence to security policies and industry standards, and real-time monitoring and incident response to detect and mitigate security threats dynamically.

By embedding security into every stage of the software development lifecycle, Green Pace ensures that security is not an afterthought but a fundamental part of development, leading to resilient and compliant software solutions.

### 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 | Likely | Medium | High | 3 |
| STD-003-CPP | High | Likely | Medium | High | 3 |
| STD-004-CPP | High | Highly Likely | Medium | High | 5 |
| STD-005-CPP | High | Likely | High | High | 4 |
| STD-006-CPP | Low | Unlikely | Low | Low | 2 |
| STD-007-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | High | Highly Likely | High | High | 5 |
| STD-010-CPP | Medium | Likely | 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 of stored data ensures that unauthorized access does not compromise sensitive information. Green Pace uses AES-256 for database and file encryption. |
| Encryption in flight | Data transmitted over networks is encrypted using TLS 1.3 to prevent interception and tampering. |
| Encryption in use | Runtime encryption protects active processes using technologies like Intel SGX to secure sensitive computations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Users must authenticate via multi-factor authentication (MFA) before accessing resources. |
| Authorization | Role-based access control (RBAC) ensures that users only have permissions necessary for their role. |
| Accounting | System logs track all user actions, including login attempts, data modifications, and administrative changes. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 01/26/2025 | Define Coding Standards | Quinlin MacKenzie | [Insert text.] |
| 1.2 | 02/16/2025 | Outline Threat Levels and Security Automation/Tools | Quinlin MacKenzie | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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