**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 | **All data received from external sources should be considered untrusted until validated. Validating input ensures that the data conforms to expected formats and boundaries, helping to prevent injection attacks, buffer overflows, and unexpected behavior. Failure to validate input is one of the most common root causes of software vulnerabilities.** |
| 1. Heed Compiler Warnings | **Compiler and static analysis warnings often identify potentially dangerous or undefined behavior in code. Developers should treat these warnings as errors and resolve them before deploying software to production. Doing so helps ensure that potential bugs or vulnerabilities are not silently ignored during development.** |
| 1. Architect and Design for Security Policies | **Security must be embedded into the software architecture from the outset, not added on later. Design decisions should reflect defined security requirements and account for likely threat models and misuse cases. A secure foundation makes future development, testing, and auditing more effective and less costly.** |
| 1. Keep It Simple | **Complex systems are harder to understand, maintain, and secure than simple ones. Simpler, cleaner code reduces the risk of errors and helps both developers and auditors identify problems more easily. Following this principle improves security posture and development efficiency.** |
| 1. Default Deny | **Systems should deny all access by default and only grant access to explicitly allowed users or services. This reduces the risk of unauthorized access due to misconfiguration or oversight. The default-deny principle is critical for creating secure access control policies.** |
| 1. Adhere to the Principle of Least Privilege | **Each user, process, or system component should operate with the minimum privileges necessary to perform its tasks. Limiting access helps reduce the damage that can be caused by a compromised or malfunctioning component. This principle is essential for limiting the blast radius of any potential breach.** |
| 1. Sanitize Data Sent to Other Systems | **Just as input must be validated, output must be sanitized to ensure it doesn't introduce risk to downstream systems. Unsanitized data can lead to injection vulnerabilities, corrupted logs, or unsafe rendering in web applications. Ensuring clean output protects system integrity and user safety.** |
| 1. Practice Defense in Depth | **Security should be enforced at multiple layers, such as application, system, and network, to protect against a variety of threats. If one control fails, others should still provide protection. Defense in depth makes systems more resilient and less dependent on a single point of failure.** |
| 1. Use Effective Quality Assurance Techniques | **Quality assurance practices, including code reviews, static analysis, and security testing, are essential for identifying issues early in the development cycle. Automated and manual testing both play a role in reducing vulnerabilities. Investing in strong QA up front saves time and reduces risks later.** |
| 1. Adopt a Secure Coding Standard | **Secure coding standards, such as those provided by SEI CERT, promote safe, consistent development practices. Following a standard helps prevent common errors that lead to security flaws. It also facilitates training, review, and auditing across development teams.** |

### 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 | Choosing the correct data types prevents unintended data truncation, loss of precision, and overflows. This improves type safety and makes the code easier to reason about and verify. |

| **Noncompliant Code** |
| --- |
| Uses a signed integer for array indexing, which can cause logic errors or vulnerabilities when negative values are incorrectly accepted. |
| int index = getArrayIndex();  if (index >= 0 && index < arrayLength) {  process(array[index]);  } |

| **Compliant Code** |
| --- |
| Uses an unsigned integer to accurately represent a non-negative array index, eliminating unnecessary checks and reducing risk. |
| unsigned int index = getArrayIndex();  if (index < arrayLength) {  process(array[index]);  } |

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

| **Principles(s):** 1 (Validate Input Data), 10 (Adopt a Secure Coding Standard): Valid data types ensure input is processed correctly and prevent security vulnerabilities related to misinterpretation of input. Secure coding standards reinforce type-safe practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 15.0 | cppcoreguidelines-type | Identifies inappropriate or unsafe use of fundamental types. |
| Coverity | 2023.6 | BAD\_ASSIGN | Detects type mismatches or dangerous assignments. SonarQube |
| SonarQube | 9.9 | c:S2142 | Flags bad conversions or implicit casts. |
| Cppcheck | 2.12 | uninitStructMember | Warns of uninitialized variables related to type safety. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validating input data values ensures that values fall within acceptable ranges. This guards against logic flaws, resource abuse, and denial-of-service conditions. |

| **Noncompliant Code** |
| --- |
| Accepts unchecked user input and passes it directly into a hardware-related function, possibly exceeding safe thresholds. |
| int speed = getUserInput();  setMotorSpeed(speed); |

| **Compliant Code** |
| --- |
| Validates the user-supplied value to ensure it falls within defined operational boundaries before proceeding. |
| int speed = getUserInput();  if (speed >= 0 && speed <= MAX\_SPEED) {  setMotorSpeed(speed);  } |

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

| **Principles(s):** 1 (Validate Input Data), 9 (Use Effective QA Techniques): Validating value ranges supports correct logic execution and prevents abuse. QA tools can help identify missing or incorrect validations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | c:ValueRangeCheck | Flags dangerous or unchecked input values. |
| Coverity | 2023.6 | CONSTANT\_EXPRESSION | Detects boundary value issues in comparisons. |
| clang-tidy | 15.0 | readability-misleading-indentation | Ensures value comparisons are not masked. |
| Polyspace | R2024a | MISRA-14.1 | Validates range limits and enforces defensive checks. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Improper handling of string buffers leads to buffer overflows and memory corruption. This standard requires bounded string operations and proper null termination. |

| **Noncompliant Code** |
| --- |
| Uses an unsafe C-style function **strcpy**, which can overflow the destination buffer with unchecked input. |
| char dest[10];  strcpy(dest, userInput); |

| **Compliant Code** |
| --- |
| Uses **strncpy** to safely copy a bounded number of characters and ensures the result is null-terminated. |
| char dest[10];  strncpy(dest, userInput, sizeof(dest) - 1);  dest[9] = '\0'; |

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

| **Principles(s):** 7 (Sanitize Data Sent to Other Systems), 10 (Adopt a Secure Coding Standard): Ensures safe handling and encoding of output, avoiding injection or memory risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 23.1 | BufferOverflow | Detects unsafe use of string copy/manipulation functions. |
| Coverity | 2023.6 | STRING\_OVERFLOW | Identifies out-of-bounds operations on buffers. |
| Cppcheck | 2.12 | bufferAccessOutOfBounds | Checks static buffer boundaries. |
| SonarQube | 9.9 | c:S3518 | Warns about dangerous use of string functions like **strcpy**. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Building SQL statements with unvalidated input allows malicious users to manipulate queries. The use of parameterized statements helps isolate logic from data. |

| **Noncompliant Code** |
| --- |
| Concatenates user input into a raw SQL query, which enables injection vulnerabilities and unauthorized access. |
| std::string query = "SELECT \* FROM users WHERE name='" + userInput + "'";  db.execute(query); |

| **Compliant Code** |
| --- |
| Implements a parameterized query that binds input securely, eliminating injection vectors. |
| PreparedStatement stmt = db.prepare("SELECT \* FROM users WHERE name=?");  stmt.bind(1, userInput);  stmt.execute(); |

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

| **Principles(s):** 5 (Default Deny), 7 (Sanitize Data Sent to Other Systems): Structured queries enforce access restrictions and data sanitization protects against injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL | Latest | sql-injection | Tracks user input through SQL query construction paths. |
| Fortify | 23.1 | SQLInjection | Detects unsafe string building and execution of SQL queries. |
| SonarQube | 9.9 | c:S3649 | Flags concatenated SQL strings with user input. |
| Checkmarx | 2023.2 | Query\_Build | Identifies unvalidated input used in query generation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Improper memory handling leads to use-after-free errors, memory leaks, and heap corruption. This standard promotes safe deallocation and pointer nullification. |

| **Noncompliant Code** |
| --- |
| Deletes memory but retains a dangling pointer, risking undefined behavior if accessed afterward. |
| double\* data = new double[100];  delete[] data;  process(data); // unsafe access |

| **Compliant Code** |
| --- |
| Deletes the allocated memory and resets the pointer, preventing accidental reuse. |
| double\* data = new double[100];  delete[] data;  data = nullptr; |

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

| **Principles(s):** 2 (Heed Compiler Warnings), 8 (Practice Defense in Depth): Compiler diagnostics and memory safety tools are layered controls that prevent low-level faults. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | Memcheck | Detects invalid memory access and use-after-free issues. |
| AddressSanitizer | LLVM 15 | heap-buffer-overflow | Finds runtime heap corruption and overflows. |
| Coverity | 2023.6 | USE\_AFTER\_FREE | Detects access to freed memory and double frees. |
| clang-tidy | 15.0 | cert-dcl30-c | Identifies unsafe pointer reuse and mismanagement. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are useful for catching logic errors during development but should not be used to validate user input. Production builds may disable assertions entirely, making them unreliable for runtime security. |

| **Noncompliant Code** |
| --- |
| Uses **assert()** to validate user input, which may be disabled and bypassed in release builds. |
| assert(userInput >= 0);  process(userInput); |

| **Compliant Code** |
| --- |
| Performs a runtime input check to enforce validation and gracefully handle failure cases. |
| if (userInput < 0) {  handleError();  } else {  process(userInput);  } |

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

| **Principles(s):** 4 (Keep It Simple), 9 (Use Effective QA Techniques): Assertions should be replaced by actual runtime logic; QA helps enforce proper boundaries and behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.6 | ASSERT\_SIDE\_EFFECT | Flags assertions that include side effects or improper conditions. |
| clang-tidy | 15.0 | misc-static-assert | Detects inappropriate static assertions in production logic. |
| SonarQube | 9.9 | c:S4031 | Identifies missing runtime checks that should replace assertions. |
| Cppcheck | 2.12 | assertWithSideEffect | Warns of assert usage with operational impact. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Failure to catch exceptions may lead to program crashes or uncontrolled behavior. Well-structured error handling ensures system robustness and fault tolerance. |

| **Noncompliant Code** |
| --- |
| Calls a function that may throw an exception but does not provide any way to handle or log errors. |
| processFile("data.txt"); |

| **Compliant Code** |
| --- |
| Catches exceptions using a **try-catch** block to log and recover from runtime errors. |
| try {  processFile("data.txt");  } catch (const std::exception& e) {  logError(e.what());  recover();  } |

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

| **Principles(s):** 3 (Architect and Design for Security Policies), 4 (Keep It Simple): Structured and scoped exception handling reduces undefined behavior and increases reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 15.0 | misc-throw-by-value | Detects improper exception usage patterns. |
| Coverity | 2023.6 | UNCAUGHT\_EXCEPTIONS | Flags unhandled exceptions in logic. |
| CodeQL | Latest | exception-leak | Identifies exception leakage across boundary layers. |
| SonarQube | 9.9 | c:S2486 | Warns when exceptions are thrown but not caught within a logical scope. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Integer Overflow** | STD-008-CPP | Unchecked arithmetic operations can cause overflows that corrupt logic or enable exploits. This standard requires use of overflow-safe operations or validation. |

| **Noncompliant Code** |
| --- |
| Performs unchecked integer addition that could silently wrap around. |
| int total = a + b; |

| **Compliant Code** |
| --- |
| Uses compiler-provided built-in function to detect and handle overflow before proceeding. |
| int total;  if (\_\_builtin\_add\_overflow(a, b, &total)) {  handleOverflow();  } else {  process(total);  } |

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

| **Principles(s):** 1 (Validate Input Data), 9 (Use Effective QA Techniques): Overflow can silently corrupt logic; defensive checks and testing help catch high-risk calculations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 15.0 | cert-int34-c | Detects integer overflows, underflows, and conversions. |
| Cppcheck | 2.12 | integerOverflow | Finds numeric overflow problems. |
| Coverity | 2023.6 | NEGATIVE\_RETURNS | Warns about unintended signed wraparound. |
| SonarQube | 9.9 | c:S2123 | Flags mathematical expressions prone to overflow. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **File I/O Handling** | STD-009-CPP | Unsafe file operations can result in data corruption or loss. Validating file streams ensures safe interaction with the filesystem. |

| **Noncompliant Code** |
| --- |
| Writes to a file without checking if it opened successfully, risking silent failure. |
| std::ofstream file("data.txt");  file << "Important data";  file.close(); |

| **Compliant Code** |
| --- |
| Confirms the file is open before writing, providing safer and more predictable behavior. |
| std::ofstream file("data.txt");  if (file.is\_open()) {  file << "Important data";  file.close();  } |

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

| **Principles(s):** 6 (Least Privilege), 9 (Use Effective QA Techniques): File operations must be verified and controlled to reduce unauthorized access and ensure correctness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | c:S3802 | Detects file streams that are not validated for success. |
| Coverity | 2023.6 | UNCHECKED\_RETURN | Flags file and system calls that lack return value checks. |
| Cppcheck | 2.12 | fileAccess | Warns when file APIs are used without checking if open succeeded. |
| clang-tidy | 15.0 | misc-io-error | Detects unhandled file operation errors. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Resource Cleanup** | STD-010-CPP | Forgetting to release resources leads to leaks and unstable runtime behavior. Use of RAII or smart pointers ensures deterministic cleanup. |

| **Noncompliant Code** |
| --- |
| Allocates memory and returns early without freeing, leading to a memory leak. |
| char\* buffer = new char[1024];  if (!initialize(buffer)) {  return;  }  process(buffer);  delete[] buffer; |

| **Compliant Code** |
| --- |
| Uses a smart pointer to automatically clean up memory, even on early return. |
| std::unique\_ptr<char[]> buffer(new char[1024]);  if (!initialize(buffer.get())) {  return;  }  process(buffer.get()); |

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

| **Principles(s):** 8 (Defense in Depth), 10 (Adopt a Secure Coding Standard): Automatic cleanup ensures consistent behavior and reduces the risk of leaks or dangling resources. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 15.0 | cppcoreguidelines-owning-memory | Detects improper memory/resource ownership and encourages smart pointers. |
| Valgrind | 3.21 | Memcheck | Finds leaks due to improper cleanup. |
| Coverity | 2023.6 | RESOURCE\_LEAK | Detects missing **delete** or **close** calls for acquired resources. |
| Cppcheck | 2.12 | leakReturnValNotUsed | Warns when function return values like **fopen()** are not cleaned up properly. |

### 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 will be used throughout the DevSecOps pipeline to enforce and maintain compliance with Green Pace’s ten secure coding standards. In the pre-production phase, static analysis tools such as **clang-tidy**, **SonarQube**, and **Coverity** are integrated into the build stage to automatically scan for policy violations at every commit. During the design and planning stages, secure architecture modeling and threat modeling tools can flag risks early in the lifecycle. Furthermore, automated testing during the verify stage uses tools like Valgrind, CodeQL, and Fortify to ensure compliance with memory safety, input validation, and resource handling rules. In the production phase, configuration validation, SIEM logging, and intrusion detection tools such as Splunk and OSSEC provide continuous monitoring for violations or suspicious behavior. Additionally, automated response systems are configured to block malicious activity, roll back changes, and alert security personnel if a breach or policy failure occurs. Overall, these automation practices ensure continuous policy enforcement, reduce human error, and uphold the organization’s defense-in-depth and Triple-A security strategies.

### 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 | Low | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | Critical | Likely | Medium | Very High | 5 |
| STD-005-CPP | High | Likely | High | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | Medium | Possible | Medium | Medium | 3 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | Medium | Possible | Low | Medium | 3 |
| STD-010-CPP | High | Likely | Low | High | 4 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at Rest | Data must be encrypted when stored on disk, in backups, or in databases. Green Pace uses AES-256 encryption for all rest-state data. This protects against theft or exposure if the storage medium is compromised. |
| Encryption in Flight | Data must be encrypted while being transmitted between systems, services, or clients. Green Pace uses TLS 1.3 or higher to secure APIs, internal service calls, and third-party integrations. This protects against man-in-the-middle attacks and eavesdropping. |
| Encryption in Use | Sensitive data processed in memory must be protected using secure enclaves or in-memory encryption mechanisms. This applies in high-security contexts such as financial, health, or classified systems. It defends against memory scraping or runtime breaches. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies user identity before access is granted. Green Pace enforces multi-factor authentication and unique credentials via OAuth2 or SSO platforms. All services must authenticate system-to-system access via tokens or certificates. |
| Authorization | Governs what authenticated users are allowed to do. Green Pace uses Role-Based Access Control with the principle of least privilege. Access is reviewed monthly, and any elevation of privileges must be audited and approved. |
| Accounting | Tracks and logs actions performed by users and systems. All access to sensitive data, config changes, and user-level changes are logged and reviewed periodically. Logs are retained for at least 12 months in tamper-resistant storage. |

**\***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 | 07/14/2025 | Partial Template Completion | Brayden Cook | Brayden Cook |
| 2.0 | 08/05/2025 | Full Template Completion | Brayden Cook | Brayden Cook |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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