**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 | Ensuring that all input data is validated and prevents security vulnerabilities such as buffer overflows, SQL injection, and cross-site scripting. This principle enforces strict input validation techniques to mitigate risks from malicious input. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential security vulnerabilities. By treating warnings as errors and resolving them proactively, developers can eliminate problematic code prior to deployment. |
| 1. Architect and Design for Security Policies | Security must be an integral part of software design. Secure architecture ensures proper implementation of security controls such as authentication, authorization, and encryption from the initial development stages. |
| 1. Keep It Simple | Simplicity reduces the attack surface of an application. Complex designs and unnecessary dependencies introduce vulnerabilities, making the application harder to secure and maintain. |
| 1. Default Deny | Access control should follow a default deny policy, where only explicitly authorized users and operations are allowed. This principle minimizes unintended access to sensitive resources. |
| 1. Adhere to the Principle of Least Privilege | Each user and process should operate with the minimum privileges necessary to perform its functions. This reduces the potential impact of security breaches. |
| 1. Sanitize Data Sent to Other Systems | Data shared between different systems must be sanitized to prevent injection attacks and ensure compatibility with intended security mechanisms. |
| 1. Practice Defense in Depth | Multiple layers of security controls provide redundancy, making it more difficult for attackers to breach a system. Combining network security, endpoint protection, and software hardening enhances resilience. |
| 1. Use Effective Quality Assurance Techniques | Security vulnerabilities can be detected and mitigated through rigorous quality assurance measures, including code reviews, penetration testing, and static analysis tools. |
| 1. Adopt a Secure Coding Standard | Using secure coding standards such as SEI CERT C++ ensures best practices are followed to minimize vulnerabilities in the code. |

### 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 correct use of data types to prevent type confusion and memory corruption. |

| **Noncompliant Code** |
| --- |
| Allocating memory for a character buffer but treating it as an integer pointer can lead to type-related memory issues. |
| int \*ptr = (int \*)malloc(sizeof(char) \* 10); |

| **Compliant Code** |
| --- |
| Ensures the allocated memory is correctly sized for the intended data type, preventing memory corruption. |
| int \*ptr = (int \*)malloc(sizeof(int) \* 10); |

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

| **Principles(s):**   * **Principle 1: Validate Input Data**   + Ensure that data used in programs is checked for proper type and format before being processed, preventing errors and vulnerabilities like type confusion or memory corruption. * **Principle 2: Heed Compiler Warnings**   + Compiler warnings related to data type issues, such as mismatched types or conversions, must be treated as errors to prevent vulnerabilities. * **Principle 9: Use Effective QA Techniques**   + Through code review and static analysis tools, issues with data type usage can be detected early to ensure correct and safe handling of data.   **Justification:** These principles ensure that data types are used correctly, validated, and not mismanaged in a way that could lead to memory corruption or security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v9.9 LTS | cpp:S3655 | Detects incorrect memory allocation or misuse of data types. |
| Cppcheck | v2.13 | typeMismatch | Detects type conversion mismatches. |
| Clang Static Analyzer | v15.0 | core.DynamicTypePropagation | Detects invalid casts and unsafe use of types. |
| Coverity | v2024.03 | RETURN\_LOCAL | Warns of incorrect return of local type references. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Prevents use of uninitialized or incorrect data values. |

| **Noncompliant Code** |
| --- |
| Using an uninitialized variable can lead to unpredictable behavior and security vulnerabilities. |
| int x;  printf(“%d”, x); |

| **Compliant Code** |
| --- |
| Initializing variables before use ensures predictable behavior and eliminates undefined behavior. |
| int x = 0;  printf(“%d”, x); |

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

| **Principles(s):**   * **Principle 1: Validate Input Data**   + Validates that data values are properly checked before being used, preventing issues such as uninitialized or incorrect data values which could lead to undefined behavior. * **Principle 9: Use Effective QA Techniques**   + Employs techniques like static analysis and testing to catch any potential issues with data values, ensuring they are properly initialized and handled.   **Justification:** The principle of validating input data helps ensure that only properly initialized and valid data is processed, which supports the prevention of issues related to incorrect or uninitialized values. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | v15.0 | cppcoreguidlines-init-variables | Warns on use of uninitialized variables. |
| SonarQube | v9.9 LTS | cpp:S1172 | Detects unused parameters/variables. |
| Coverity | v2024.03 | UNINIT | Flags uninitialized variables. |
| Polyspace Bug Finder | ? | R2024a | Detects undefinable variable values and control paths. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Prevents buffer overflows and improper string handling. |

| **Noncompliant Code** |
| --- |
| Copying a string without checking its length can cause buffer overflows, leading to memory corruption. |
| char str[5];  strcpy(str, “TooLong”); |

| **Compliant Code** |
| --- |
| Using strncpy with explicit size constraints and null-termination prevents buffer overflow. |
| char str[6];  strncpy(str, “Short”, 5);  str[5] = ‘\0’; |

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

| **Principles(s):**   * **Principle 1: Validate Input Data**   + Properly validates strings before they are processed, ensuring that buffer overflows and improper string handling is prevented. * **Principle 3: Architect and Design for Security Policies**   + String handling should be designed to be secure from the outset, enforcing safe coding practices for dealing with strings to prevent vulnerabilities. * **Principle 7: Sanitize Data Sent to Other Systems**   + Ensures that strings shared with other systems are sanitized to prevent injection attacks or incompatible data from being processed.   **Justification:** The principle of validating input data helps ensure that only properly initialized and valid data is processed, which supports the prevention of issues related to incorrect or uninitialized values. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Low | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | v2024.03 | BUFFER\_SIZE | Detects buffer overflow. |
| Fortify SCA | v23.1 | Buffer Overflow | Detects unsafe string functions. |
| SonarQube | v9.9 LTS | cpp:S3518 | Identifies potential buffer overflows. |
| Clang Static Analyzer | v15.0 | security.insecureAPI.strcpy | Flags use of unsafe string handling APIs. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevents SQL injection attacks by using parameterized queries. |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries exposes the system to SQL injection attacks. |
| String query = “SELECT \* FROM users WHERE name=’” + userInput + “’”; |

| **Compliant Code** |
| --- |
| Using parameterized queries ensures that user input is treated as data, not executable code. |
| PreparedStatement \*stmt = conn->prepareStatement(“SELECT \* FROM users WHERE name=?”);  Stmt->setString(1, userInput); |

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

| **Principles(s):**   * **Principle 1: Validate Input Data**   + Prevents malicious input in SQL queries, such as unescaped characters, by validating data before it is processed, thereby preventing SQL injection. * **Principle 7: Sanitize Data Sent to Other Systems**   + Ensures that data sent to databases is sanitized, particularly when constructing SQL queries, preventing any malicious injection attacks like SQL injection.   **Justification:** By validating and sanitizing input data used in SQL queries, these principles ensure that the application is protected from SQL injection attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | v23.1 | SQL Injection | Flags raw SQL with user input. |
| SonarQube | v9.9 LTS | cpp:S2077 | Detects injection risks from unsanitized input. |
| Veracode | v2024.1 | SQL Injection | Detects dynamic SQL construction flaws. |
| Checkmarx | CxSAST | SQL Injection | Identifies query construction issues. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Prevents memory leaks and unauthorized access. |

| **Noncompliant Code** |
| --- |
| Allocating memory without a deallocation mechanism can result in memory loss. |
| char \*buffer = new char[10]; |

| **Compliant Code** |
| --- |
| Using smart pointers ensures proper memory management and automatic deallocation. |
| std::unique\_ptr<char[]> buffer(new char[10]); |

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

| **Principles(s):**   * **Principle 4: Keep It Simple**   + A simpler memory model reduces the complexity of memory management, making it easier to track and protect memory allocations. * **Principle 6: Adhere to the Principle of Least Privilege**   + Ensures that memory access is restricted to the minimum required by the application, reducing potential attack vectors for unauthorized access.   **Justification:** Simplicity and the least privilege principle help minimize the complexity of memory handling and reduce the potential for unauthorized memory access or leaks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Low | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | v2.13 | Memleak | Detects memory leaks. |
| Valgrind | v3.21 | Memcheck | Detects memory errors/leaks. |
| Clang AddressSanitizer | ? | Heap-buffer-overflow | Runtime detection of memory misuse. |
| Coverity | v2024.03 | RESOURCE\_LEAK | Identifies memory/resource management issues. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Ensures code correctness by validating assumptions. |

| **Noncompliant Code** |
| --- |
| Assertions should not be used for input validation in production code, as they may be disabled. |
| int x = -1;  assert(x > 0); |

| **Compliant Code** |
| --- |
| Using exception handling properly ensures robust error handling without unintended termination. |
| if (x <= 0)  throw std::invalid\_argument(“Invalid input”); |

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

| **Principles(s):**   * **Principle 2: Heed Compiler Warnings**   + Compiler warnings related to assertions or assumptions in code should be treated as errors to ensure that invalid assumptions are caught before deployment. * **Principle 9: Use Effective QA Techniques**   + Assertions are part of ensuring code correctness and can be validated during static analysis and testing, ensuring that assumptions hold true and prevent bugs.   **Justification:** Assertions ensure that assumptions made in code are correct, supporting code correctness and security by preventing faulty logic. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | v9.9 LTS | cpp:S3519 | Avoid inappropriate assertions. |
| Coverity | v2024.03 | ASSERT\_SIDE\_EFFECT | Detects logical misuse of assertions. |
| Clang-Tidy | v15.0 | cert-dcl03-c | Warns against using assertions for runtime checks. |
| Fortify | ? | Assertion Violation | Warns about assumptions that can fail silently in production. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Improves error handling by using exceptions instead of error codes. |

| **Noncompliant Code** |
| --- |
| Returning error codes can lead to inconsistent error handling. |
| if (ptr == NULL)  return -1; |

| **Compliant Code** |
| --- |
| Throwing exceptions ensures that errors are propagated and handled correctly. |
| if (ptr == nullptr)  throw std::runtime\_error(“Memory allocation failed”); |

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

| **Principles(s):**   * **Principle 8: Practice Defense in Depth**   + Proper exception handling adds an additional layer of defense, ensuring that unexpected errors are properly managed to prevent cascading failures and vulnerabilities. * **Principle 9: Use Effective QA Techniques**   + Exception handling can be treated as part of quality assurance, ensuring that errors are properly handled without compromising system security or stability.   **Justification:** Exception handling improves error management and supports the overall security posture by managing errors that could otherwise expose vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | v15.0 | core.CallAndMessage | Detects missed error handling. |
| SonarQube | v9.9 LTS | cpp:S3512 | Ensures exceptions are properly handled. |
| Coverity | v2024.03 | UNCAUGHT\_EXCEPT | Detects unhandled exceptions. |
| Cppcheck | v2.13 | throwInDestructor | Warns about dangerous exception throwing in destructors. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | [STD-008-CPP] | Prevents race conditions and deadlocks in multithreaded applications. |

| **Noncompliant Code** |
| --- |
| Accessing shared variables without synchronization can lead to race conditions. |
| int shared;  void update() { shared++; } |

| **Compliant Code** |
| --- |
| Using mutexes ensures thread-safe access to shared resources. |
| std::mutex m;  int shared;  void update() { std::lock\_guard<std::mutex> lock(m); shared++; } |

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

| **Principles(s):**   * **Principle 6: Adhere to the Principle of Least Privilege**   + Threads should be given the least privilege necessary, ensuring that no unnecessary permissions are granted that could increase the attack surface. * **Principle 8: Practice Defense in Depth**   + Thread safety is a critical part of defense in depth, as race conditions or deadlocks can be exploited by attackers. Thread safety ensures that concurrent access is controlled and secure.   **Justification:** Ensuring thread safety and reducing unnecessary thread privileges prevents race conditions and deadlocks, supporting both system stability and security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Low | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helgrind | Valgrind v3.21 | Thread race detection | Identifies race conditions. |
| Clang ThreadSanitizer | ? | data-race | Detects concurrency issues. |
| Coverity | v2024.03 | GUARD\_CONDITION | Flags unsafe shared variable access. |
| SonarQube | v9.9 LTS | cpp:S2333 | Detects non-thread-safe statics and shared data access. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | [STD-009-CPP] | Avoids dangling pointers and memory corruption. |

| **Noncompliant Code** |
| --- |
| Accessing a pointer after deletion results in undefined behavior and potential crashes. |
| int \*ptr = new int(5);  delete ptr;  \*ptr = 10; |

| **Compliant Code** |
| --- |
| Using smart pointers ensures safe memory management and prevents dangling pointers. |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(5); |

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

| **Principles(s):**   * **Principle 2: Heed Compiler Warnings**   + Compiler warnings related to pointer safety (i.e., null pointers or dereferencing invalid addresses) should be treated as errors to prevent memory corruption. * **Principle 4: Keep It Simple**   + Reducing pointer complexity minimizes errors related to pointer safety, simplifying memory management and preventing dangling pointers or memory corruption.   **Justification:** Pointer safety is directly supported by validating pointer usage and ensuring simplicity in pointer logic to avoid vulnerabilities like memory corruption or unauthorized access. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Low | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | LLVM v15 | Use-after-free detection |  |
| Coverity | v2024.03 | BAD\_FREE | Detects invalid/dangling pointer operations. |
| Valgrind | v3.21 | Memcheck | Identifies improper memory reuse. |
| Cppcheck | v2.13 | nullPointer | Detects null dereference and pointer misuse. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Cryptography | [STD-010-CPP] | Ensures secure data encryption and hashing. |

| **Noncompliant Code** |
| --- |
| Using weak or outdated cryptographic functions exposes data to security risks. |
| string hash = md5(password); |

| **Compliant Code** |
| --- |
| Using strong cryptographic algorithms like bcrypt ensures secure password storage. |
| std::string hash = bcrypt::generateHash(password); |

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

| **Principles(s):**   * **Principle 3: Architect and Design for Security Policies**   + Cryptography should be designed and implemented from the start, ensuring that encryption and hashing are properly integrated into the software’s architecture to meet security requirements. * **Principle 8: practice Defense in Depth**   + Cryptography serves as a key layer in defending sensitive data from unauthorized access, ensuring that data is encrypted even if other security layers fail.   **Justification:** Designing cryptography into the architecture from the start and using multiple layers of security protects sensitive data against breaches and ensures robust data confidentiality and integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Low | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Veracode | v2024.1 | Insecure Cryptographic Storage | Warns of deprecated algorithms. |
| Fortify | ? | Weak Crypto | Detects use of outdated or broken ciphers. |
| Checkmarx | CxSAST | Cryptographic Issues | Flags hardcoded keys or weak implementations. |
| SonarQube | v9.9 LTS | cpp:S5542 | Warns against insecure hash functions and key management. |

### 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 embedded in Green Pace’s CI/CD pipeline using tools like SonarQube, Coverity, and Cppcheck. These tools will be triggered during the code commit, build, and test stages of the DevSecOps pipeline. Rules from SEI CERT C++ and CWE top vulnerabilities will be checked automatically. Any violations will halt the build process and push issues to a tracking system such as Jira or GitHub Issues. Developers must fix issues before code is approved for deployment, ensuring continuous compliance. Additionally, Git hooks can enforce style and secure code pre-commit, and periodic security scans will run in the staging environment for compliance validation.

### 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 | Probable | Medium | P12 | L1 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | High | Probable | Low | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Probable | Low | P18 | L1 |
| STD-006-CPP | Medium | Unlikely | Low | P6 | L2 |
| STD-007-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-008-CPP | High | Probable | Low | P18 | L1 |
| STD-009-CPP | High | Probable | Low | P18 | L1 |
| STD-010-CPP | High | Probable | Low | P18 | L1 |

### 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 | Encrypts data stored on disk using **AES-256** or equivalent algorithms. Applies to databases, backups, and logs to protect sensitive information from theft or unauthorized access. Required for all systems storing personally identifiable or proprietary data. |
| Encryption in flight | Ensures secure data transmission using **TLS 1.3** or newer. Applies to all network communications between services, clients, and APIs to protect data from eavesdropping or man-in-the-middle (MITM) attacks. |
| Encryption in use | Protects data currently being processed in memory. Enforced via trusted execution environments (TEE) or memory encryption (i.e., AMD SEV or Intel SGX). Applicable to applications handling highly sensitive data such as financial or healthcare information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies user identities using Multi-Factor Authentication (MFA) and Single Sign-On (SSO). Enforced at login portals and API access points. Prevents unauthorized access. |
| Authorization | Grants access based on Role-Based Access Control (RBAC). Users only gain access to systems and files necessary for their roles. Limits the scope of potential damage in case of a breach. |
| Accounting | Tracks user activity using centralized logging (i.e., SIEM tools). Captures login attempts, data access, changes to records, and admin actions. Supports incident response and audits. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 03/22/2025 | Updated coding standards with examples of good and bad code, added 3 additional standards (Milestone 3-2). | Corey Sampson |  |
| 1.2 | 04/10/2025 | Added risk assessments, automation tools, justification of principles involved with the 10 coding standards, and policies for encryption and Triple A (Project One). | Corey Sampson |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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