**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 entering the system must be checked for type, length, format, and range before use. By validating inputs at the earliest opportunity, we prevent malicious or malformed data from propagating through the application and causing SQL injections, buffer overflows, or logic errors. Validation should be performed both client and server side. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential issues such as uninitialized variables, type mismatches, or unreachable code that can lead to undefined behavior. Developers should treat warnings as errors or resolve them promptly to maintain code correctness and prevent latent security vulnerabilities |
| 1. Architect and Design for Security Policies | Security requirements must be considered during system architecture and design, not bolted on after implementation. This principle ensures that threat modeling, trust boundaries, and defense in depth layers guide development decisions from the outset, leading to robust and maintainable security controls. |
| 1. Keep It Simple | Complex code is harder to understand, review, and test, leading to a higher risk of security bugs. By minimizing unnecessary features and dependencies, the codebase remains clear, making it easier to spot vulnerabilities and reduce the attack surface. |
| 1. Default Deny | All access and operations should be denied by default, granting permissions only when explicitly authorized. This approach ensures that new code paths or features do not inadvertently expose privileged functionality or data. |
| 1. Adhere to the Principle of Least Privilege | Components and users should operate with the minimal privileges necessary to perform their functions. Limiting privileges reduces the potential impact of a compromised component and confines security breaches to the smallest possible scope. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to other systems or components, such as databases, logging frameworks, or external services, proper sanitization and encoding must be applied to prevent injection attacks and ensure data integrity across trust boundaries. |
| 1. Practice Defense in Depth | Implement multiple, redundant layers of security controls. Such as input validation, authentication, authorization, encryption so that if one control fails, others still protect the system. Defense in depth increases resilience against a variety of attack vectors. |
| 1. Use Effective Quality Assurance Techniques | Incorporate static analysis, dynamic testing, and peer reviews into the development lifecycle. Automated tools should detect known coding defects, while manual reviews uncover design level and logic flaws that tools might miss. |
| 1. Adopt a Secure Coding Standard | Formalize best practices for safe coding in the organization by defining and enforcing a secure coding standard. Consistent application of these rules ensures that all developers follow the same guidelines, reducing variance and potential security gaps. |

### 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** | **Use Fixed-Width and Checked Numeric Types** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Ensuring variables use appropriately sized types (e.g., int32\_t) and checking for overflows prevents integer truncation, wraparound, and precision loss that can lead to logic errors or buffer overruns. |

| **Noncompliant Code** |
| --- |
| Implicit narrowing conversion from larger to smaller integer type without validation. |
| // Noncompliant: may truncate 64-bit value to 32-bit  long largeCount = getRecordCount();  int count = largeCount; |

| **Compliant Code** |
| --- |
| Use fixed-width type and check for range before assignment. |
| // Compliant: explicit check prevents overflow  int64\_t largeCount = getRecordCount();  if (largeCount > std::numeric\_limits<int32\_t>::max()) {  throw std::overflow\_error("Record count too large");  }  int32\_t count = static\_cast<int32\_t>(largeCount); |

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

| **Principles(s):** Adopt a secure coding standard (Policy 10): Following a formal secure coding standard (e.g., CERT C++) mandates consistent type selection and conversion rules across the codebase |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | cppcoreguidelines-narrowing-conversions | sign changes |
| Cppcheck | 2.12 | warning (narrowing/truncation) | Finds implicit conversions and integer truncation risks |
| SonarQube (C/C++) | 10.4 | Conversion/overflow rules | Quality gate on dangerous implicit conversions/overflows |
| UBSan (clang) | 17.0 | UndefinedBehaviorSanitizer | Runtime checks for signed overflow/UB in tests |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Enforce Range and Format Checks** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating that numeric and formatted data fall within acceptable boundaries prevents out-of-range values from triggering unexpected behavior or security issues. |

| **Noncompliant Code** |
| --- |
| Accepts user input without range checks. |
| int age;  std::cin >> age; // no check on input value |

| **Compliant Code** |
| --- |
| Check that input is within expected range. |
| int age;  if (!(std::cin >> age) || age < 0 || age > 150) {  throw std::invalid\_argument("Invalid age provided");  } |

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

| **Principles(s):** Validate input data (P1): Validate all inputs and indexes (bounds checks, strict parse/format verification) to stop out-of-range access and format mismatches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | boundscheck | Detects-out-of-bounds element access |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | bugprone-narrowing-conversions | Catches unsafe parses/unchecked conversions |
| |  | | --- | | SonarQube (C/C++) |  |  | | --- | |  | | 10.4 | Input validation/format rules | Enforces validation of user input and format strings |
| |  | | --- | | libFuzzer (clang) |  |  | | --- | |  | | 17.0 | Fuzz targets | Finds bad ranges/parse paths at runtime |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Prefer std::string over C-Style Buffers** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Using std::string helps manage memory automatically and provides boundary checks, reducing buffer overflow risks associated with fixed-size char arrays. |

| **Noncompliant Code** |
| --- |
| Fixed-size buffer without length check. |
| char name[32];  strcpy(name, userInput); // unsafe copy |

| **Compliant Code** |
| --- |
| Use std::string and copy safely |
| std::string name = userInput; // dynamic sizing |

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

| **Principles(s):** Keep it simple (P4): Using std::string/std::string\_view instead of raw char\* buffers simplifies memory management and eliminates many overflow/underflow risks |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | unsafeFunctions | Flags strcpy/strcat & buffer errors |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | modernize-use-std-string | Nudges to safer C++ strings |
| SonarQube  (C/C++) | 10.4 | Buffer/unsafe API rules | Blocks unsafe C string APIs |
| |  | | --- | | AddressSanitizer |  |  | | --- | |  | | 17.0 | ASan | Runtime detection of over/under-runs |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use Parameterized Queries and ORM Interfaces** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Building SQL statements through concatenation enables injection attacks. Parameterized queries ensure data is treated as values, not executable code. |

| **Noncompliant Code** |
| --- |
| Concatenates user input into SQL string |
| std::string query = "SELECT \* FROM users WHERE name='" + userName + "'";  executeSQL(query); |

| **Compliant Code** |
| --- |
| Use parameter marker and bind variables |
| auto stmt = db.prepare("SELECT \* FROM users WHERE name=?");  stmt.bind(1, userName);  stmt.execute(); |

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

| **Principles(s):** Sanitize data sent to other systems (P7): bind parameters or use an ORM so the database receives typed values rather than concatenated strings—effectively sanitizing outbound data to the DB. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Possible | Low | Critical | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Semgrep | 1.64 | generic.sql-injection | Finds string-built queries; enforces bind params |
| |  | | --- | | SonarQube  (C/C++) |  |  | | --- | |  | | 10.4 | SQL injection rules | Static detection + quality gate |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | clang-analyzer-security | Taints/sinks for unsafe use |
| OWASP ZAP (DAST) | 2.14 | Active scan | Runtime injection probes on staging |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Check All Heap and Stack Allocations** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Verifying successful memory allocations and bounds prevents null pointer dereference and buffer overflow vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Uses malloc without checking return |
| char\* buffer = (char\*)malloc(size);  strcpy(buffer, data); |

| **Compliant Code** |
| --- |
| Validate allocation and use safe copy |
| [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):** Use effective QAT (P10): Following secure coding standards formalizes required checks and RAII patterns across the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | memleak, nullPointer, resourceLeak | Leak/null checks |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | cppcoreguidelines-owning-memory | Ownership & allocation patterns |
| SonarQube (C/C++) | 10.4 | Resource/NULL handling rules | Ensures allocation results are checked |
| ASan/UBSan/LSan | 17.0 | Sanitizers | Runtime out-of-bounds/UB/leak detection |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Deploy Assertions for Detectable Programmer Errors** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions document invariants and catch logic errors during development. They should not replace error handling for invalid user input. |

| **Noncompliant Code** |
| --- |
| No assertion on critical invariant |
| int divisor = getDivisor();  int result = dividend / divisor; |

| **Compliant Code** |
| --- |
| Assert non-zero divisor. |
| int divisor = getDivisor();  assert(divisor != 0);  int result = dividend / divisor; |

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

| **Principles(s):** Architect & design for security (P3): Assertions document and enforce invariants/pre-conditions at design time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17.0 | deadcode, core | Unreachable/contradictory conditions |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | readability-assertion-side-effect | Safe assertion usage |
| SonarQube (C/C++) | 10.4 | Defensive programming rules | Verifies pre/post conditions patterns |
| |  | | --- | | GTest (CI) |  |  | | --- | |  | | lastest | Debug builds w/ asserts | Fails tests when invariants break |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Catch Specific Exceptions and Clean Up Resource** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catching all exceptions or leaving resources open can hide errors and cause resource leaks. Handling only relevant exception types ensures intended recovery. |

| **Noncompliant Code** |
| --- |
| Catch-all handler masks errors |
| try {  riskyOperation();  } catch (...) {  // silent  } |

| **Compliant Code** |
| --- |
| Catch and log specific exception, then clean up |
| try {  riskyOperation();  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what();  cleanup();  throw;  } |

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

| **Principles(s):** Keep it simple (P4): Catching specific exception types and handling them near the source keeps control flow predictable and easy to reason about. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | except | Catch-all misuse, rethrow issues |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | hicpp-exception-baseclass | Exception safety & hierarchy |
| |  | | --- | | SonarQube (C/C++) |  |  |  | | --- | --- | |  |  | | 10.4 | Error handling rules | Enforces specific catches/cleanup paths |
| |  | | --- | | AddressSanitizer |  |  | | --- | |  | | 17.0 | ASan in tests | Confirms no leaks on exception paths |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Prefer RAII for All Resources** |
| --- | --- | --- |
| **Resource Lifetime Management** | [STD-008-CPP] | Using RAII classes (ex. std::unique\_ptr, std::lock\_guard) ensures that resources such as memory, file handles, and locks are released automatically, preventing leaks. |

| **Noncompliant Code** |
| --- |
| Manual delete may be skipped on exception |
| Widget\* w = new Widget();  // ... code that may throw  delete w; // never reached if exception |

| **Compliant Code** |
| --- |
| Use unique\_ptr for automatic cleanup |
| std::unique\_ptr<Widget> w = std::make\_unique<Widget>();  // ... code that may throw  // no need to delete explicitly |

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

| **Principles(s):** Keep it simple (P4): RAII ties resource lifetime to scope so acquisition and cleanup are automatic and easy to read. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | cppcoreguidelines-raii | Enforces RAII patterns |
| |  | | --- | | Cppcheck |  |  | | --- | |  | | 2.12 | resourceLeak | Detects missing frees/closes |
| |  | | --- | | SonarQube (C/C++) |  |  | | --- | |  | | 10.4 | Resource lifetime rules | Blocks manual new/free patterns |
| |  | | --- | | Valgrind |  |  | | --- | |  | | 3.22 | memcheck | Runtime leak/use-after-free detection |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use Thread-Safe Synchronization Primitives** |
| --- | --- | --- |
| **Secure Concurrency** | [STD-009-CPP] | Concurrent access to shared data must be protected with mutexes or atomic operations to prevent data races and inconsistent state that attackers can exploit. |

| **Noncompliant Code** |
| --- |
| Unsynchronized shared counter |
| int counter = 0;  void increment() { counter++; } |

| **Compliant Code** |
| --- |
| Use atomic type for thread safety |
| std::atomic<int> counter{0};  void increment() { counter.fetch\_add(1, std::memory\_order\_relaxed); } |

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

| **Principles(s):** Architect & design for security (P3): Concurrency is a design concern choose the right primitives (std::mutex, std::atomic) and avoid shared-mutable state. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17.0 | Concurrency checks (concurrency-mt-unsafe) | Flags MT-unsafe APIs & atomics misuse |
| |  | | --- | | SonarQube (C/C++) |  |  | | --- | |  | | 10.4 | Concurrency rules | Race-prone patterns & locking misuse |
| |  | | --- | | Cppcheck |  |  | | --- | |  | | 2.12 | threadsafety | Basic thread-safety heuristics |
| ThreadSanitizer | 17.0 | TSan | Runtime race/deadlock detection in CI |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use Proven Libraries and Avoid Hardcoded Keys** |
| --- | --- | --- |
| **Cryptographic Hygiene** | [STD-010-CPP] | Implementing cryptography incorrectly can introduce vulnerabilities. Use vetted libraries like OpenSSL and manage keys via secure stores rather than embedding them in code. |

| **Noncompliant Code** |
| --- |
| Hardcoded encryption key |
| const std::string key = "MySecretKey123";  encrypt(data, key); |

| **Compliant Code** |
| --- |
| Retrieve key from secure vault |
| std::string key = SecureVault::getKey("encryption");  encrypt(data, key); |

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

| **Principles(s):** Least privilege (P6): Give every user/service only the minimum access needed default deny, then narrowly allow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Gitleaks | 8.18 | Default ruleset | Scans commits/PRs for secrets |
| Semgrep | 1.64 | security.hardcoded-credentials, weak-crypto rules | Finds hardcoded keys and weak algorithms |
| SonarQube (C/C++) | 10.4 | Secrets/crypto rules | Quality gate on secret usage/weak ciphers |
| OWASP Dependency-Check (or Snyk) | 9.0 | NVD CVE match | Flags vulnerable crypto libraries |

### 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.

[Insert your written explanations here.]

### 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 | Possible | |  | | --- | | Low |  |  | | --- | |  | | High | 4 |
| |  | | --- | | **STD-003-CPP** |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | |  | | --- | | Possible |  |  | | --- | |  | | Low | High | 4 |
| |  | | --- | | **STD-004-CPP** |  |  | | --- | |  | | |  | | --- | | Critical |  |  | | --- | |  | | |  | | --- | | Possible |  |  | | --- | |  | | Low | Critical | 5 |
| |  | | --- | | **STD-005-CPP** |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | Possible | Medium | High | 4 |
| |  | | --- | | **STD-006-CPP** |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | Unlikely | Low | Medium | 2 |
| |  | | --- | | **STD-007-CPP** |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | Unlikely | Low | High | 3 |
| |  | | --- | | **STD-008-CPP** |  |  | | --- | |  | | High | Possible | Low | High | 4 |
| |  | | --- | | **STD-009-CPP** |  |  | | --- | |  | | High | Possible | Medium | High | 4 |
| |  | | --- | | **STD-010-CPP** |  |  | | --- | |  | | Critical | Possible | Medium | Critical | 5 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest ensures that all stored data (databases, backups, files, and logs) is encrypted using strong algorithms such as AES-256. This protects data if physical media (servers, drives, or cloud storage) are stolen or accessed without authorization. The policy applies at all times to sensitive and personal information stored by the organization. |
| Encryption in flight | Encryption in flight secures data as it travels across networks by requiring TLS (Transport Layer Security) or equivalent encryption protocols. This prevents interception, tampering, or eavesdropping on sensitive communications between users, services, and external systems. The policy applies to all internal and external communication channels, APIs, and user logins. |
| Encryption in use | Encryption in use ensures that sensitive data remains encrypted even during active processing in memory, often using techniques like homomorphic encryption or trusted execution environments. This protects against memory scraping or insider threats. The policy applies when handling highly sensitive data (such as authentication credentials, encryption keys, or financial information). |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that only verified users and systems can access company resources. Strong authentication methods, such as multi-factor authentication (MFA) and single sign-on (SSO), are required for user logins, remote access, and administrative accounts. This policy prevents unauthorized access. |
| Authorization | Authorization enforces least-privilege access by granting users only the rights necessary to perform their role. Role-based access control (RBAC) or attribute-based access control (ABAC) ensures that database changes, file access, and administrative actions are limited to approved roles. This prevents privilege misuse or escalation. |
| Accounting | Accounting ensures all user actions and system changes are logged and monitored. Logs track user logins, file access, database changes, and new user additions. This supports auditing, forensic investigations, and compliance requirements. The policy applies continuously and is enforced by centralized logging and monitoring systems. |

**\***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.

Coding Standard 1 – Input Validation

* Principle(s): 1 (Least Privilege), 6 (Fail Securely)
* Validating input ensures only safe data is processed, preventing privilege escalation through malformed input. Enforcing strict validation also supports failing securely, because invalid input is rejected rather than processed in an unsafe way.

Coding Standard 2 – Enforce Range and Format Checks

* Principle(s): 1 (Least Privilege), 7 (Separation of Duties)
* Range checks protect against out-of-range values that may alter privileges or corrupt data. By applying separation of duties, data handling is segmented so no single error compromises the system.

Coding Standard 3 – Error and Exception Handling

* Principle(s): 6 (Fail Securely), 10 (Defense in Depth)
* Proper error handling ensures that failures don’t leak sensitive details or create security gaps, aligning with fail securely. Defense in depth is supported because even if one check fails, layered exception handling reduces risk.

Coding Standard 4 – Resource Management

* Principle(s): 2 (Defense in Depth), 8 (Economy of Mechanism)
* Properly managing memory and resources ensures that no single failure leads to compromise. Keeping mechanisms simple and efficient reduces the attack surface.

Coding Standard 5 – Secure Data Storage

* Principle(s): 3 (Complete Mediation), 5 (Least Common Mechanism)
* All access to stored data is mediated and checked every time, ensuring secure use. Shared resources are minimized to reduce potential cross-user data leaks.

Coding Standard 6 – Access Control

* Principle(s): 1 (Least Privilege), 9 (Open Design)
* Limiting user privileges reduces exposure if credentials are compromised. Open design ensures the control mechanisms are based on well-established, tested methods rather than obscurity.

Coding Standard 7 – Secure Communication

* Principle(s): 4 (Open Design), 10 (Defense in Depth)
* Secure protocols that are openly reviewed help ensure trustworthiness. Using encryption and authentication adds layers of defense, preventing interception or tampering.

Coding Standard 8 – Logging and Monitoring

* Principle(s): 3 (Complete Mediation), 6 (Fail Securely)
* Logging ensures all access attempts are mediated and recorded, providing evidence of misuse. Failures are logged securely without exposing sensitive details.

Coding Standard 9 – Secure Configuration

* Principle(s): 2 (Defense in Depth), 8 (Economy of Mechanism)
* Secure defaults and layered protections prevent misconfiguration from leading to vulnerabilities. Simplicity reduces complexity and makes secure configurations easier to maintain.

Coding Standard 10 – Code Review and Testing

* Principle(s): 7 (Separation of Duties), 9 (Open Design)
* Having multiple developers review code ensures separation of duties and reduces insider risk. Open design supports using standard, tested review practices rather than relying on secrecy.

**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 |  |
| [Insert text.] | 07/28/2025 | Added coding standards 1–10 and risk assessment details | Jose Munoz | Inshan Singh, Mr. (M.Sc) |
| [Insert text.] | 08/18/2025 | Completed encryption & Triple-A framework policies, automation tools, and final principles mapping | Jose Munoz | Inshan Singh, Mr. (M.Sc) |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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