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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Always check and confirm that any data entered into a system is correct, expected, and safe. This prevents malicious inputs, like SQL injection or buffer overflows, that can compromise system security. |
| 1. Heed Compiler Warnings | Pay attention to and resolve all warnings from the compiler, as they often point to potential bugs or vulnerabilities that could be exploited if left unchecked. |
| 1. Architect and Design for Security Policies | Security should be integrated into the design phase of a system. This means building systems with clear security policies, such as access controls and encryption protocols, from the ground up. |
| 1. Keep It Simple | Simple code and architecture are easier to test, understand, and secure. Complex systems increase the risk of hidden vulnerabilities and make maintenance harder. |
| 1. Default Deny | Systems should deny access by default and only allow what is permitted. This minimizes the chance of unauthorized access or privilege escalation. |
| 1. Adhere to the Principle of Least Privilege | Each user or component should only have the minimum level of access necessary to perform its function, reducing the potential damage from compromised accounts or processes. |
| 1. Sanitize Data Sent to Other Systems | Clean and verify data before sending it to other systems to prevent injection attacks or corrupting external databases or services. |
| 1. Practice Defense in Depth | Use multiple layers of security such as firewalls, authentication, and encryption so that if one layer fails, others still protect the system. |
| 1. Use Effective Quality Assurance Techniques | Regular testing, code reviews, and security audits help identify and fix issues early, improving the overall security and reliability of software. |
| 1. Adopt a Secure Coding Standard | Follow established guidelines for writing secure code to ensure consistency, reduce vulnerabilities, and make it easier for teams to maintain secure practices throughout the development lifecycle. |

### 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 visually distinct identifiers** |
| --- | --- | --- |
| **Data Type** | [STD-001-C++] | Using clear, distinct identifiers prevents confusion between similarly named variables and helps avoid logic errors in code. |

| **Noncompliant Code** |
| --- |
| Uses visually similar names that may cause confusion. |
| int l = 1;  int I = 2;  int result = l + I; // easy to misread |

| **Compliant Code** |
| --- |
| Uses distinct and descriptive variable names. |
| int lower = 1;  int upper = 2;  int result = lower + upper; |

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

| **Principles(s):** 4 & 10: Simpler, clearer variable names reduce logic errors and make code easier to maintain and audit. A secure coding standard ensures naming rules are consistently followed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | Inconclusive-naming | Warns about naming convention violations |
| Clang-Tidy | 17 | Readability-identifer-naming | Flags poorly named or confusing identifiers |
| SonarQube | 10 | Cpp:S100 | Enforces naming conventions |
| Visual Studio Analyzer | 2022 | CodeStyle | Enforces code style and naming rules |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that operations on signed integers do not result in overflow** |
| --- | --- | --- |
| **Data Value** | [STD-002-C++] | Signed integer overflow leads to undefined behavior, which can compromise program stability and security. |

| **Noncompliant Code** |
| --- |
| Integer addition results in overflow. |
| int a = INT\_MAX;  int b = 1;  int c = a + b; // overflow |

| **Compliant Code** |
| --- |
| Checks for overflow before performing addition. |
| int a = INT\_MAX;  int b = 1;  if (a <= INT\_MAX - b) {  int c = a + b;  } |

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

| **Principles(s):** 1, 5, & 9: Validating arithmetic boundaries prevents unsafe operations. Deny overflows by default and detect via QA tools like sanitizers and static analyzers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 17 | -fsanitize=undefined | Detects integer overflow |
| Coverity | 2024.1 | INT\_OVERFLOW | Static analysis of overflow risks |
| Fortify SCA | 23.2 | Integer Overflow | Identifies overflow conditions |
| CodeSonar | 7.2 | Integer Overflow | Finds arithmetic overflow problems |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-C++] | Modifying string literals is undefined behavior and may crash the program. |

| **Noncompliant Code** |
| --- |
| Attempts to modify a string literal. |
| char\* str = "hello";  str[0] = 'H'; // undefined behavior |

| **Compliant Code** |
| --- |
| Uses a character array instead of a string literal. |
| char str[] = "hello";  str[0] = 'H'; // safe |

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

| **Principles(s):** 3, 7, & 10: Enforces memory safety by treating string literals as immutable. Design-time rules help avoid violations, and a secure standard prevents accidental misuse. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17 | clang-analyzer-cplusplus.StringLiteralModification | Detects modification of string literals |
| Cppcheck | 2.9 | stringModification | Warns when string literals are modified |
| CodeSonar | 7.2 | StringLiteralWrite | Detects writes to read-only memory |
| Visual Studio Analyzer | 2022 | C26451 | Warns about potential writes to immutable data |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C++] | SQL injection vulnerabilities can allow attackers to manipulate queries and access sensitive data. |

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

| **Compliant Code** |
| --- |
| Uses parameterized queries to prevent SQL injection. |
| PreparedStatement ps = connection.prepareStatement("SELECT \* FROM users WHERE name = ?");  ps.setString(1, user\_input);  ps.executeQuery(); |

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

| **Principles(s):** 1, 7, 3, & 8: Input validation and output sanitization are crucial to prevent SQLi. Design systems to use parameterized queries and layer protections like input filters. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Medium | Critical | 0 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 23.2 | SQL Injection | Detects unvalidated database access |
| SonarQube | 10 | cpp:S2077 | Detects SQL injection patterns |
| Checkmarx | 2024 | SQLi | Identifies unsafe SQL concatenation |
| Semgrep | 1.47 | c.lang.security.sql-injection | Pattern matching for vulnerable SQL |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C++] | Accessing freed memory leads to undefined behavior and potential exploits. |

| **Noncompliant Code** |
| --- |
| Accesses a pointer after it has been deleted. |
| int\* ptr = new int(10);  delete ptr;  \*ptr = 5; // use-after-free |

| **Compliant Code** |
| --- |
| Nullifies the pointer after deletion. |
| int\* ptr = new int(10);  delete ptr;  ptr = nullptr; |

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

| **Principles(s):** 6, 8, & 10: Prevents use-after-free by restricting access and layering runtime checks. Secure coding rules enforce pointer nullification after deletion. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Medium | High | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | memcheck | Detects use-after-free at runtime |
| AddressSanitizer | LLVM 17 | UAF | Detects invalid memory access |
| CodeSonar | 7.2 | UseAfterFree | Static detection of UAF |
| Coverity | 2024.1 | USE\_AFTER\_FREE | Identifies post-deletion access |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Detect and remove code that has no effect or is unreachable** |
| --- | --- | --- |
| **Assertions** | [STD-006-C++] | Unreachable or ineffective code can mask logic errors and hinder maintainability. |

| **Noncompliant Code** |
| --- |
| Contains code that will never execute. |
| assert(false);  std::cout << "This line will never execute"; // unreachable |

| **Compliant Code** |
| --- |
| Removes unreachable code and uses assert meaningfully. |
| if (conditionFails()) {  assert(false && "Unexpected condition occurred");  } |

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

| **Principles(s):** 4 & 9: Simpler code is easier to understand and test. Removing unreachable code eliminates ambiguity and potential hidden bugs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17 | deadcode.DeadStores | Finds dead code |
| Cppcheck | 2.9 | unreachableCode | Warns about unreachable code |
| Coverity | 2024.1 | DEADCODE | Detects code paths that never execute |
| SonarQube | 10 | cpp:S1871 | Identifies and flags redundant code |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Do not abruptly terminate the program** |
| --- | --- | --- |
| **Exceptions** | [STD-007-C++] | Unexpected termination through exit or abort disrupts program control and hinders error handling. |

| **Noncompliant Code** |
| --- |
| Uses abort without cleanup. |
| if (critical\_error) {  std::abort(); // harsh termination  } |

| **Compliant Code** |
| --- |
| Uses exception handling to manage critical errors. |
| if (critical\_error) {  throw std::runtime\_error("Critical failure");  } |

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

| **Principles(s):** 3, 8, & 9: Enables graceful error handling through exception structures rather than immediate termination, ensuring predictability and testability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17 | misc-exception-usage | Warns about poor exception practices |
| SonarQube | 10 | cpp:S3512 | Warns against abrupt termination |
| Coverity | 2024.1 | BAD\_TERMINATION | Flags use of exit/abort |
| Visual Studio Analyzer | 2022 | C26442 | Suggests structured error handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Avoid double free** |
| --- | --- | --- |
| Memory | [STD-008-C++] | Freeing memory more than once leads to undefined behavior and can be exploited. |

| **Noncompliant Code** |
| --- |
| Deletes the same pointer twice. |
| int\* p = new int;  delete p;  delete p; // double delete |

| **Compliant Code** |
| --- |
| Sets pointer to null after deletion. |
| int\* p = new int;  delete p;  p = nullptr; |

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

| **Principles(s):** 5, 6, 8, & 10: Resources are tightly managed access is denied by default, and runtime or static layers detect multiple deallocations. Consistent coding practices prevent recurrence. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Medium | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | LLVM 17 | double-free | Runtime detection of double free |
| Valgrind | 3.21 | memcheck | Detects invalid frees |
| CodeSonar | 7.2 | DoubleFree | Finds memory freed more than once |
| Coverity | 2024.1 | DOUBLE\_FREE | Flags double deallocation |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not use magic numbers** |
| --- | --- | --- |
| Data | [STD-009-C++] | Magic numbers reduce code readability and maintainability. |

| **Noncompliant Code** |
| --- |
| Uses a hardcoded number in logic. |
| if (speed > 88) {  timeTravel();  } |

| **Compliant Code** |
| --- |
| Replaces magic number with a named constant. |
| const int MAX\_SPEED = 88;  if (speed > MAX\_SPEED) {  timeTravel();  } |

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

| **Principles(s):** 4 & 10: Named constants clarify the intent of code and simplify maintenance. Secure coding standards promote consistency across the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17 | readability-magic-numbers | Flags hardcoded numbers |
| Cppcheck | 2.9 | magicNumber | Warns about unexplained constants |
| SonarQube | 10 | cpp:S109 | Promotes use of named constants |
| Visual Studio Analyzer | 2022 | CodeStyle | Enforces modern style guides |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Validate user input** |
| --- | --- | --- |
| Input Validation | [STD-010-C++] | Unvalidated input can lead to crashes or security breaches. |

| **Noncompliant Code** |
| --- |
| Uses input without checking. |
| int age;  std::cin >> age;  processAge(age); // no validation |

| **Compliant Code** |
| --- |
| Checks input value range before processing. |
| int age;  std::cin >> age;  if (age >= 0 && age <= 120) {  processAge(age);  } |

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

| **Principles(s):** 1, 7, 8, & 9: Validating and sanitizing input is foundational. Input checks prevent malicious use, and testing ensures input logic is correctly enforced. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Medium | Critical | 0 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 23.2 | Input Validation | Detects lack of input sanitation |
| SonarQube | 10 | cpp:S2674 | Detects lack of input validation |
| Checkmarx | 2024 | InputSanitization | Flags unvalidated user input |
| Semgrep | 1.47 | c.lang.security.input-validation | Pattern-matches vulnerable input use |

### 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.

To enforce the secure coding standards defined in this policy, automation should be embedded throughout Green Pace’s existing DevOps process. The DevSecOps model illustrated in the diagram divides the software lifecycle into two major domains: Pre-production and Production. Security must be treated as a shared responsibility across development, security, and operations teams, and automation is key to ensuring continuous compliance.

In the Assess and Plan phase of pre-production, teams evaluate threat landscapes, regulatory updates, and project requirements. At this stage, Green Pace can incorporate automated policy-as-code checks that evaluate design decisions against security baselines. Coding standards like using visually distinct identifiers and validating input (Standards 1 and 10) should be embedded in requirement templates and planning checklists to ensure secure design from the outset.

During Design and Build, developers can use IDE-based static analysis tools such as SonarLint or Clang-Tidy to receive real-time feedback about violations, including unsafe memory operations or the use of magic numbers (Standards 2, 3, 5, 8, 9). This also includes integrating compilers with hardening flags like -Wall, -Wextra, and -fsanitize=address to detect dangerous patterns at build time. Compliant and noncompliant patterns should be validated by code linters and commit hooks to prevent insecure code from reaching the repository.

The Verify and Test stage automates the use of fuzz testing, dynamic analysis, and vulnerability scanners. Tools like Valgrind, AFL, and AddressSanitizer can detect input validation issues, use-after-free errors, and unhandled exceptions (Standards 5, 7, 10). Integration with continuous integration (CI) systems ensures that only code passing these tests is allowed to progress to production.

As code transitions into Production, the Transition and Health Check phase allows automation to verify configurations and deploy hardened builds. Here, enforcement of coding standards continues with infrastructure-as-code validation, automated penetration testing, and runtime instrumentation. For instance, validation of error handling routines (Standard 7) ensures the system responds gracefully to critical failures.

In the Monitor and Detect phase, logs and alerts from tools like SIEM (Security Information and Event Management) systems can detect anomalies tied to unsafe code paths or logic errors that evaded earlier enforcement. This phase supports standards such as "Remove unreachable code" (Standard 6) and "Do not abruptly terminate the program" (Standard 7) by monitoring their effects during execution.

Lastly, Respond and Maintain and Stabilize provide feedback loops. If coding standard violations are detected—such as double frees or SQL injection attempts (Standards 4 and 8)—automated response mechanisms can trigger rollback scripts, alert engineering teams, and restore baseline configurations. This ensures rapid containment and guides continuous improvement.

### 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 | Low | Medium | Low | Moderate | 3 |
| STD-002-CPP | High | Medium | Medium | High | 1 |
| STD-003-CPP | High | High | Low | High | 1 |
| STD-004-CPP | Critical | High | Medium | Critical | 0 |
| STD-005-CPP | Critical | Medium | High | High | 1 |
| STD-006-CPP | Medium | Medium | Low | Moderate | 2 |
| STD-007-CPP | High | Low | Low | Moderate | 2 |
| STD-008-CPP | Critical | Medium | Medium | High | 1 |
| STD-009-CPP | Low | High | Low | Moderate | 3 |
| STD-010-CPP | Critical | High | Medium | Critical | 0 |

### 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 | **What it is**: Encryption at rest protects stored data using cryptographic techniques. This includes files, databases, and backups. **How and Why**: This policy requires that all sensitive data (e.g., personally identifiable information, financial data) stored on physical or virtual storage systems be encrypted using AES-256 or higher. Disk encryption tools like BitLocker (Windows) or LUKS (Linux) must be enabled on all endpoints and servers. **When**: Applies at all times data is saved to disk, cloud storage, or archived. Protects against data theft in case of physical compromise or improper disposal of storage media. |
| Encryption in flight | **What it is**: Encryption in flight secures data as it travels across networks using protocols such as TLS (Transport Layer Security). **How and Why**: This policy mandates that all internal and external communications—between applications, APIs, and users—must use encrypted channels. HTTPS must be enforced on all web services, and emails containing sensitive information must use S/MIME or PGP encryption. **When**: Applies to all data transferred over wired or wireless networks, including remote access, API calls, and mobile communication. This protects against eavesdropping and man-in-the-middle attacks. |
| Encryption in use | **What it is**: Encryption in use refers to protecting data while it is actively being processed in memory or by the CPU. **How and Why**: This policy requires implementing techniques like trusted execution environments (TEEs), homomorphic encryption, or memory encryption (e.g., Intel SGX or AMD SEV) for sensitive processing tasks. **When**: Applies to operations involving confidential or proprietary data processed on endpoints or cloud environments. Especially critical in multi-tenant cloud platforms or when handling high-value computations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | |  | | --- | |  |  |  | | --- | | **What it is**: Authentication verifies the identity of users or systems before access is granted. **How and Why**: All systems must use multi-factor authentication (MFA) to verify user identity. Authentication logs must be maintained and monitored. Login attempts must be rate-limited to prevent brute-force attacks. **When**: Applied at every login, session start, and access attempt to ensure only valid users gain entry. Mandatory for accessing systems, APIs, or sensitive internal resources. | |
| Authorization | **What it is**: Authorization defines what authenticated users are allowed to do or access. **How and Why**: This policy requires the use of role-based access control (RBAC). Each user should only have the minimum level of access necessary (principle of least privilege). Administrative privileges should be reviewed quarterly. The addition of new users must be approved by a supervisor and tracked in an access control system. **When**: Applied after authentication during access decisions, file permissions, and administrative operations. Prevents overreach and data misuse. |
| Accounting | |  | | --- | |  |  |  | | --- | | **What it is**: Accounting logs and tracks user activity, providing audit trails for actions taken on systems or data. **How and Why**: This policy requires comprehensive logging of user logins, file access, database changes, and administrative actions. Logs must be centralized in a SIEM solution (e.g., Splunk, ELK) and retained for a minimum of 12 months. All actions must be traceable to a specific user ID. **When**: Applied continuously across all environments—production, staging, and development—to support compliance, incident response, and forensics. | |

**\***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

\*\*See Principles section under each standard for the Mapping of Principles to each standard as well as the explanation.\*\*

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 | 05/23/2025 | Completed Principles and 10 Standards | Sean Born | [Insert text.] |
| [Insert text.] | 06/11/2025 | Completed remainder of document. | Sean Born | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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