**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 | Ensure all user inputs are validated against expected formats to prevent injection attacks and other vulnerabilities. |
| 1. Heed Compiler Warnings | Address all compiler warnings to catch potential security flaws and bugs during the development phase. |
| 1. Architect and Design for Security Policies | Develop software with security policies in mind from the beginning, ensuring robust access control and data protection mechanisms. |
| 1. Keep It Simple | Reduce complexity in design to minimize attack surfaces and make security auditing easier. |
| 1. Default Deny | Implement restrictive permissions by default and only grant access when explicitly required. |
| 1. Adhere to the Principle of Least Privilege | Limit user and process privileges to the bare minimum necessary for functionality. |
| 1. Sanitize Data Sent to Other Systems | Ensure that any data being sent to external systems is sanitized to prevent code injection or data corruption. |
| 1. Practice Defense in Depth | Layer security controls at different levels (e.g., network, application, and user authentication) for enhanced protection. |
| 1. Use Effective Quality Assurance Techniques | Implement rigorous testing methodologies such as unit tests, static analysis, and security audits to maintain software integrity. |
| 1. Adopt a Secure Coding Standard | Follow industry best practices and standardized guidelines like SEI CERT to write secure and maintainable 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) | Prevents type mismatches that can lead to vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses an int type for a large value, causing an overflow issue. |
| int x = 3000000000; |

| **Compliant Code** |
| --- |
| The compliant code uses long long to properly store the large value without overflow. |
| long long x = 3000000000LL; // Uses appropriate data type |

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

| **Principles(s):** 1, 9, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12.1 | uninitStructMember, typeMismatch | Detects unsafe or incorrect type usage, including mismatched or uninitialized types. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | (STD-002-CPP) | Prevents out-of-range errors. |

| **Noncompliant Code** |
| --- |
| The noncompliant code accesses an out-of-bounds index in an array, leading to undefined behavior. |
| int arr[5];  arr[10] = 42; |

| **Compliant Code** |
| --- |
| The compliant code ensures the index is within the valid range before accessing the array element. |
| int arr[5];  if (index >= 0 && index < 5) {  arr[index] = 42;  } |

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

| **Principles(s):** 1, 9, 10 |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-exp33-c, bugprone-branch-clone | Checks for unvalidated or out-of-range values causing logic flaws or overflows. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | (STD-003-CPP) | Prevents buffer overflows. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses strcpy without checking buffer size, causing a buffer overflow. |
| char buf[5];  strcpy(buf, "TooLong"); |

| **Compliant Code** |
| --- |
| The compliant code uses strncpy with a size limit and manually null-terminates the string to prevent overflow. |
| char buf[5];  strncpy(buf, "Safe", sizeof(buf)-1);  buf[4] = '\0'; |

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

| **Principles(s):** 1, 7, 9, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.6 | PW.NEGATIVE\_SUBSCRIPT, STRING\_NULL\_TERMINATOR | Identifies off-by-one errors and missing null terminators in strings. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | (STD-004-CPP) | Prevents SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| The noncompliant code directly concatenates user input into an SQL query, making it vulnerable to injection attacks. |
| string query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| The compliant code uses parameterized queries to safely pass user input, preventing SQL injection. |
| stmt->executeQuery("SELECT \* FROM users WHERE name = ?", userInput); |

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

| **Principles(s):** 1, 7, 8, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 23.1 | SQL Injection | Flags unvalidated input passed directly into SQL queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | (STD-005-CPP) | Prevents memory leaks and undefined behavior. |

| **Noncompliant Code** |
| --- |
| The noncompliant code dynamically allocates memory but fails to release it, leading to a memory leak. |
| int\* ptr = new int[10]; |

| **Compliant Code** |
| --- |
| The compliant code uses std::unique\_ptr, ensuring proper memory management and preventing leaks. |
| std::unique\_ptr<int[]> ptr(new int[10]); |

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

| **Principles(s):** 2, 6, 8, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | memcheck | Detects buffer overflows, illegal memory access, and freed memory use. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | (STD-006-CPP) | Ensures runtime validation |

| **Noncompliant Code** |
| --- |
| The noncompliant code relies on assert, which may be disabled in release mode, leading to potential runtime errors. |
| int val = getValue();  assert(val > 0); |

| **Compliant Code** |
| --- |
| The compliant code explicitly checks the condition and throws an exception if it fails, ensuring proper error handling. |
| if (val <= 0) {  throw std::runtime\_error("Invalid value");  } |

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

| **Principles(s):**  9 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.24 | V547, V614 | Warns when assertion conditions are always true/false or unreachable. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | (STD-007-CPP) | Encourages proper exception handling. |

| **Noncompliant Code** |
| --- |
| The noncompliant code performs division without checking for zero, leading to potential runtime errors. |
| int divide(int a, int b) {  return a / b;  } |

| **Compliant Code** |
| --- |
| The compliant code checks for division by zero and throws an exception if the denominator is zero. |
| int divide(int a, int b) {  if (b == 0) throw std::invalid\_argument("Divide by zero");  return a / b;  } |

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

| **Principles(s):** 8, 10 |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0 | cert-err52-cpp | Verifies exception safety and resource cleanup consistency. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Secure File Handling** | (STD-008-CPP) | Prevents unauthorized file access. |

| **Noncompliant Code** |
| --- |
| The noncompliant code opens a file without specifying access modes, potentially leading to security issues. |
| std::ifstream file("data.txt"); |

| **Compliant Code** |
| --- |
| The compliant code explicitly sets the access mode, reducing the risk of unauthorized access. |
| std::ifstream file("data.txt", std::ios::in | std::ios::binary); |

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

| **Principles(s):** 6, 8, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Semgrep | 1.41.0 | c.lang.security.file-permissions | Detects improper file permission handling, unsafe file paths. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Pointer Safety** | (STD-009-CPP) | Prevents dangling pointers. |

| **Noncompliant Code** |
| --- |
| The noncompliant code deletes a pointer and then dereferences it, causing undefined behavior (dangling pointer). |
| int\* ptr = new int(5);  delete ptr;  \*ptr = 10; |

| **Compliant Code** |
| --- |
| The compliant code uses std::unique\_ptr, which ensures proper memory management and avoids 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):** 2, 6, 8, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.6 | NULL\_DEREF, USE\_AFTER\_FREE | Finds unsafe pointer dereferencing and use-after-free patterns. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Concurrency Safety** | (STD-010-CPP) | Avoids race conditions. |

| **Noncompliant Code** |
| --- |
| The noncompliant code increments a shared counter without synchronization, leading to data races. |
| int counter = 0;  void increment() { counter++; } |

| **Compliant Code** |
| --- |
| The compliant code uses std::atomic to ensure thread-safe increments without race conditions. |
| 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):** 8, 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | LLVM 15.0 | data race, mutex misuse | Detects concurrency errors such as race conditions and double locking. |

### 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.



### Pre-Production Automation Enhancements:

1. **Assess and Plan (Yellow Segment)**  
   Automation should begin during the assessment and planning phase by integrating tools that evaluate threat landscapes and detect policy violations early. These tools can automatically check code repositories for outdated dependencies, known vulnerabilities, or configuration drifts.
2. **Design (Blue Segment)**  
   Utilize security-as-code practices and templates that enforce design-time compliance. Integrate automated tools to ensure that secure design patterns are followed and policy-driven architectural decisions are adhered to.
3. **Build (Black Segment)**  
   During the build process, incorporate automated static code analysis, dependency scanning, and policy validation pipelines. These tools can block builds that do not meet compliance or contain known vulnerabilities.
4. **Verify and Test (Gray Segment)**  
   Automated test suites should include security tests (SAST, DAST, IAST) as well as compliance validation against infrastructure and data handling policies. Tools like automated penetration testing or compliance-as-code frameworks should be integrated here.

### Production Automation Enhancements:

1. **Transition and Health Check (Gray Segment)**  
   Use automated deployment validation tools to verify that the production environment adheres to defined standards before allowing “go-live”. Health checks should include automated security and compliance validation reports.
2. **Monitor and Detect (Black Segment)**  
   Deploy automated monitoring tools to continuously track security events, unusual behaviors, and policy violations in real time. Integrate these tools with SIEM systems for centralized alerting.
3. **Respond (Blue Segment)**  
   Automation can facilitate faster incident response by triggering predefined remediation scripts or rolling back non-compliant changes. This ensures rapid containment and policy enforcement.
4. **Maintain and Stabilize (Yellow Segment)**  
   Regular compliance audits and vulnerability scans can be scheduled and automated. Automated patch management tools can be used to ensure the infrastructure remains up to date with minimal manual intervention.

### Integration with Existing DevOps Process:

Key recommendations include:

* Integrating security testing and compliance checks into CI/CD pipelines.
* Defining and enforcing security policies as code using tools like Open Policy Agent (OPA), Chef InSpec, or HashiCorp Sentinel.
* Automating access controls, audit logging, and secure configuration baselines in version control.
* Ensuring visibility and reporting through centralized dashboards for compliance metrics.

### Summary of Risk Assessments

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rule | Severity | Likelihood | Remediation Cost | Priority | Level | Overall Risk (1-5) |
| STD-001-CPP | Medium | Likely | Low | Medium | 2 | 3 |
| STD-002-CPP | Medium | Likely | Medium | Medium | 2 | 3 |
| STD-003-CPP | High | Probable | Medium | High | 1 | 4 |
| STD-004-CPP | High fully | Likely | Medium | High | 1 | 5 |
| STD-005-CPP | High | Likely | Medium | High | 1 | 4 |
| STD-006-CPP | Low | Unlikely | Low | Low | 3 | 1 |
| STD-007-CPP | Medium | Probable | Medium | Medium | 2 | 3 |
| STD-008-CPP | High | Likely | High | High | 1 | 4 |
| STD-009-CPP | High | Probable | Medium | High | 1 | 4 |
| STD-010-CPP | High | Likely | High | High | 1 | 4 |

**Justification for the Risk Levels:**

* **STD-001-CPP (Data Type):**
  + Severity: Medium - Errors in data type usage can lead to logical errors or misinterpretation of data.
  + Likelihood: Likely - Incorrect data type usage can happen relatively easily, especially during input or when type casting.
  + Remediation Cost: Low - Addressing type-related issues usually involves minor code fixes and static analysis.
  + Overall Risk: 3 (Moderate).
* **STD-002-CPP (Data Value):**
  + Severity: Medium - Invalid data values can lead to unexpected program behavior or even vulnerabilities if boundary conditions are mishandled.
  + Likelihood: Likely - Data value issues are common, especially with user input or external data sources.
  + Remediation Cost: Medium - Fixing data value errors might involve input validation, error handling, and code modifications.
  + Overall Risk: 3 (Moderate)
* **STD-003-CPP (String Correctness):**
  + Severity: High - String-related vulnerabilities like buffer overflows are serious and exploitable.
  + Likelihood: Probable - String manipulation errors are relatively common, especially in C++.
  + Remediation Cost: Medium - Addressing string issues may require careful code reviews and secure coding practices.
  + Overall Risk: 4 (High)
* **STD-004-CPP (SQL Injection):**  
  \* Severity: High - SQL injection allows an attacker to execute arbitrary code within a database.
  + Likelihood: Likely - Especially if user input is used directly in SQL queries without appropriate sanitization.
  + Remediation Cost: Medium - Requires using parameterized queries and input sanitization.  
    \* Overall Risk: 5 (Very High)
* **STD-005-CPP (Memory Protection):**
  + Severity: High - Memory corruption can lead to program crashes, denial-of-service, or even arbitrary code execution.
  + Likelihood: Likely - Memory management in C++ requires meticulous attention to detail.
  + Remediation Cost: Medium - Debugging and fixing memory issues may be complex.
  + Overall Risk: 4 (High)
* **STD-006-CPP (Assertions):**
  + Severity: Low - Assertions help detect and identify programming errors.
  + Likelihood: Unlikely - Assertions are used for testing and don't typically appear in production.
  + Remediation Cost: Low - Addressing failing assertions usually involves fixing the underlying logical error.
  + Overall Risk: 1 (Very Low)
* **STD-007-CPP (Exceptions):**
  + Severity: Medium - Poor exception handling can leave a system vulnerable and unreliable.
  + Likelihood: Probable - Developers can forget to implement robust exception handling.
  + Remediation Cost: Medium - Implementing robust exception handling can require refactoring.
  + Overall Risk: 3 (Moderate)
* **STD-008-CPP (Secure File Handling):**
  + Severity: High - Improper handling of files can lead to arbitrary code execution, information disclosure, or data loss.  
    \* Likelihood: Likely - Common to have file access vulnerabilities.
  + Remediation Cost: High - Requires a solid understanding of the underlying file system and the application's file access needs.  
    \* Overall Risk: 4 (High)
* **STD-009-CPP (Pointer Safety):**  
  \* Severity: High - Pointer issues such as dangling pointers and null pointer dereferences can lead to crashes and exploitable vulnerabilities.  
  \* Likelihood: Probable - These issues can appear during the debugging process.
  + Remediation Cost: Medium - Usually requires a clear understanding of how pointers are used.  
    \* Overall Risk: 4 (High)

**STD-010-CPP (Concurrency Safety):**

* Severity: High - Race conditions and deadlocks can lead to corrupted data and application crashes.
* Likelihood: Likely - multi-threaded applications and the improper use of shared memory can cause problems.
* Remediation Cost: High - Requires a solid understanding of concurrency.
* Overall Risk: 4 (High)

### Create Policies for Encryption and Triple A

1. Encryption at Rest Policy:

* Policy Statement: All sensitive data stored on persistent storage (databases, file systems, backups) must be encrypted at rest using approved cryptographic algorithms and key management practices.
  + User Logins: User account passwords stored within databases must be encrypted using strong hashing algorithms.
  + Changes to the Database: Databases must be encrypted using approved encryption algorithms.
  + Addition of New Users: When new users are added to the system, their credentials must be encrypted prior to storage.
  + User Level of Access: Encryption key access will be restricted based on the principle of least privilege.
  + Files Accessed by Users: All sensitive files stored on the file systems must be encrypted at rest.
* Rationale: To safeguard confidential data against unauthorized access and data breaches.

2. Encryption in Flight Policy:

* Policy Statement: All sensitive data transmitted over networks (internal and external) must be encrypted in transit using approved protocols such as TLS/SSL.
  + User Logins: User authentication requests must be encrypted during transmission.
  + Changes to the Database: Database transactions involving sensitive data must be encrypted during transit.
  + Addition of New Users: The transmission of newly added user credentials should be encrypted.
  + User Level of Access: Authorization requests transmitted across networks must be encrypted.
  + Files Accessed by Users: File downloads and uploads containing sensitive data must be encrypted.
* Rationale: To protect data from eavesdropping, interception, and tampering during transmission.

3. Encryption in Use Policy:

* Policy Statement: Sensitive data processed in memory should be encrypted or masked whenever possible. If full encryption is not possible, sensitive data should be stored in memory in a secure fashion that minimizes exposure.
  + User Logins: Passwords used for authentication should be masked when processed in memory.
  + Changes to the Database: In-memory caching of sensitive data should be minimized or encrypted.
  + Addition of New Users: Credentials of new users must be handled securely in memory and masked when processed.
  + User Level of Access: Authorization policies should be loaded into memory securely.
  + Files Accessed by Users: If files are cached in memory they should be stored in a secure fashion.
* Rationale: To reduce the risk of sensitive data exposure when it's actively being processed.

4. Authentication Policy:

* Policy Statement: All users must be authenticated using strong, secure methods before being granted access to the system. Multifactor authentication is strongly encouraged for elevated privileges.
  + User Logins: Users must provide credentials to verify their identity prior to gaining access.
  + Changes to the Database: Changes to the database require authentication to proceed.
  + Addition of New Users: Administrative users responsible for adding new user accounts must be properly authenticated.
  + User Level of Access: Authentication determines which level of access a user should receive.
  + Files Accessed by Users: Access to files is based on the user's authenticated identity.
* Rationale: To ensure that only authorized users gain access to the system and its resources.

5. Authorization Policy:

* Policy Statement: Access to system resources and data must be based on the principle of least privilege. Users should be granted only the necessary permissions for their designated roles.
  + User Logins: Authentication is followed by authorization.
  + Changes to the Database: Users need authorization to perform database modification.
  + Addition of New Users: Users need appropriate administrative access to add new users.
  + User Level of Access: User level of access will be defined and enforced by authorization policies.
  + Files Accessed by Users: Authorization policies will define access levels for files based on user roles.
* Rationale: To prevent unauthorized access and modifications to the system by limiting user permissions.

6. Accounting Policy:

* Policy Statement: All access to system resources and data must be audited and logged. Audit logs should be reviewed regularly for security anomalies.
  + User Logins: All login attempts, successful and unsuccessful, must be logged.
  + Changes to the Database: All modifications to the database schema and data must be logged.
  + Addition of New Users: User creation, deletion, and modification must be logged.
  + User Level of Access: Any changes to user permissions must be logged.
  + Files Accessed by Users: All file access attempts must be logged, especially for sensitive or high-value files.
* Rationale: To enable monitoring of system activities, identify and track security violations, and provide an audit trail for forensic analysis.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to encrypting data stored on physical media (e.g., databases, file systems). Green Pace mandates full-disk encryption (FDE) and database-level encryption for all sensitive data. This ensures that even if storage devices are stolen or improperly accessed, the data remains protected. This policy applies especially to customer records, source code, and credentials. |
| Encryption in flight | This ensures data is protected while being transmitted across networks. TLS 1.2+ is required for all HTTP communications, API requests, and internal microservice communication. This policy helps defend against man-in-the-middle attacks and eavesdropping. Encryption in flight must be enforced during logins, user data entry, and API communication. |
| Encryption in use | This protects data being actively processed in memory. Techniques such as homomorphic encryption and secure enclaves (e.g., Intel SGX) are recommended where sensitive computations occur. It applies to Green Pace’s real-time analytics pipelines and any services handling decrypted sensitive data in runtime memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users or systems before access is granted. All users must authenticate using multi-factor authentication (MFA) with a minimum requirement of a strong password and one additional factor (token, biometric). Applies to all user logins, CI/CD pipelines, and admin access. This prevents unauthorized access at the entry point. |
| Authorization | Authorization defines what an authenticated user can access. Role-Based Access Control (RBAC) will be enforced, assigning privileges based on job function. All new users must be approved by security admins, and roles reviewed monthly. Applies to user level of access and file/resource permissions. |
| Accounting | |  | | --- | |  |  |  | | --- | | Accounting tracks and logs user activity such as file access, data changes, and login times. Audit logs must be immutable and retained for at least 12 months. Applies to logins, database changes, new user additions, and file access monitoring. These logs support forensic analysis, compliance, and incident response. | |

### Map the Principlesto standards

1) Data Type (STD-001-CPP):

* Principles: 1, 9, 10
  + 1 (Validate Input Data): Verifying that the input data conforms to the expected data type (e.g., integer, string) before further processing. This helps prevent type-related errors and potential vulnerabilities.
  + 9 (Use Effective Quality Assurance Techniques): Applying unit testing and code reviews to test the validity and correctness of data type usage.
  + 10 (Adopt a Secure Coding Standard): Following coding guidelines that specify how data types should be used securely, including explicit type conversions and avoiding implicit assumptions.

2) Data Value (STD-002-CPP):

* Principles: 1, 9, 10
  + 1 (Validate Input Data): Validating input data against expected ranges or allowed sets of values. This prevents unexpected or out-of-bounds values from causing issues.
  + 9 (Use Effective Quality Assurance Techniques): Employing testing techniques like boundary value analysis and fuzzing to ensure that code handles valid and invalid data values correctly.
  + 10 (Adopt a Secure Coding Standard): Adhering to standards that promote secure handling of data values, including limits and constraints for specific data fields.

3) String Correctness (STD-003-CPP):

* Principles: 1, 7, 9, 10
  + 1 (Validate Input Data): Ensuring that string inputs meet expected length and format restrictions to prevent buffer overflows or other string-related vulnerabilities.
  + 7 (Sanitize Data Sent to Other Systems): Encoding string outputs before sending them to external systems (e.g., databases, APIs) to prevent injection attacks.
  + 9 (Use Effective Quality Assurance Techniques): Utilizing fuzz testing with various string inputs to ensure resilience against malformed data.
  + 10 (Adopt a Secure Coding Standard): Adhering to a coding standard that promotes safe string manipulation, including proper null-termination, length management, and avoiding unsafe string functions.

4) SQL Injection (STD-004-CPP):

* Principles: 1, 7, 8, 10
  + 1 (Validate Input Data): Sanitizing and validating user input used in SQL queries to prevent injection attacks.
  + 7 (Sanitize Data Sent to Other Systems): Using parameterized queries or stored procedures rather than concatenating user input directly into SQL statements.
  + 8 (Practice Defense in Depth): Implementing multiple layers of defense against SQL injection, such as input validation, output encoding, and least privilege database user accounts.
  + 10 (Adopt a Secure Coding Standard): Following secure coding guidelines specifically for database interactions and SQL query construction.

5) Memory Protection (STD-005-CPP):

* Principles: 2, 6, 8, 10
  + 2 (Heed Compiler Warnings): Taking compiler warnings seriously, as they often indicate potential memory-related issues like buffer overflows.
  + 6 (Adhere to the Principle of Least Privilege): Allocating only necessary memory and freeing it properly to prevent memory leaks and vulnerabilities.
  + 8 (Practice Defense in Depth): Implementing multiple memory protection strategies like address space layout randomization (ASLR) and stack canaries.
  + 10 (Adopt a Secure Coding Standard): Adhering to a coding standard that guides safe memory management practices, such as avoiding direct memory manipulation with unsafe functions.

6) Assertions (STD-006-CPP):

* Principles: 9
  + 9 (Use Effective Quality Assurance Techniques): Implementing assertions throughout the code, especially in critical sections and function entry points, to check for logical errors and runtime contract violations.

7) Exceptions (STD-007-CPP):

* Principles: 8, 10
  + 8 (Practice Defense in Depth): Implementing robust exception handling to gracefully recover from unexpected situations and prevent exploitable crashes.
  + 10 (Adopt a Secure Coding Standard): Adhering to a standard that guides secure exception handling practices, like logging exceptions appropriately and avoiding leaking sensitive information in error messages.

8) Secure File Handling (STD-008-CPP):

* Principles: 6, 8, 10
  + 6 (Adhere to the Principle of Least Privilege): Granting file access permissions based on the principle of least privilege to prevent unauthorized access.
  + 8 (Practice Defense in Depth): Verifying file paths and permissions, implementing file integrity checks, and handling file I/O errors gracefully to minimize risk of file-related vulnerabilities.
  + 10 (Adopt a Secure Coding Standard): Following guidelines that address secure file I/O, like properly handling file paths and avoiding unsafe file functions.

9) Pointer Safety (STD-009-CPP):

* Principles: 2, 6, 8, 10
  + 2 (Heed Compiler Warnings): Paying attention to compiler warnings about potential pointer issues (e.g., dangling pointers).
  + 6 (Adhere to the Principle of Least Privilege): Only access memory through valid pointers and avoid dereferencing null or invalid pointers.
  + 8 (Practice Defense in Depth): Using smart pointers to automate memory management and reduce the risk of memory leaks or access violations.
  + 10 (Adopt a Secure Coding Standard): Adhering to a coding standard that promotes safe pointer usage, like using smart pointers and avoiding direct pointer arithmetic.

10) Concurrency Safety (STD-010-CPP):

* Principles: 8, 10
  + 8 (Practice Defense in Depth): Utilizing synchronization mechanisms (e.g., mutexes, semaphores) to prevent race conditions and ensure safe access to shared resources from multiple threads.
  + 10 (Adopt a Secure Coding Standard): Adhering to a coding standard for multithreaded programming, ensuring the safe use of shared data and avoiding concurrency issues.

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

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

## 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 |  |
| 1.5 | 03/15/2025 | Edited draft re-organization | Marcus Stanley | Marcus Stanley |
| 2.0 | 04/08/2025 | Updated Final Draft | Marcus Stanley | Marcus Stanley |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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