**Green Pace Developer: Security Policy Guide Colin Kwasnik**



# 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 external inputs must be validated to prevent malicious data from compromising system integrity. Proper validation ensures that only correctly formatted and expected data types are processed. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential code issues. By addressing them, developers can prevent many types of vulnerabilities, such as buffer overflows or data truncation. |
| 1. Architect and Design for Security Policies | Security should be integrated into the system design from the start. This means considering threats and countermeasures during the planning and architectural stages, not just at implementation. |
| 1. Keep It Simple | Complexity introduces risk. A simpler codebase is easier to understand, test, and secure, making it less prone to security flaws. |
| 1. Default Deny | Systems and applications should deny access by default. Permissions should be explicitly granted only to trusted entities, minimizing exposure to unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Users and systems should operate using the least amount of access required to perform their functions. This limits potential damage from a breach or malfunction. |
| 1. Sanitize Data Sent to Other Systems | Before passing data to external systems or APIs, it should be cleaned to remove harmful elements, preventing injection attacks or protocol manipulation. |
| 1. Practice Defense in Depth | Multiple layers of security controls should be used to protect systems. If one layer fails, others still provide protection. |
| 1. Use Effective Quality Assurance Techniques | Code should undergo thorough testing, including static analysis and code reviews, to catch vulnerabilities early in the development lifecycle. |
| 1. Adopt a Secure Coding Standard | Standardized secure coding practices, like SEI CERT, help ensure consistency and reduce common programming errors that can lead to vulnerabilities. |

### 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: Ensure correct data types are used** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using appropriate data types prevents overflows, underflows, and type conversion issues. |

| **Noncompliant Code** |
| --- |
| Improper use of signed and unsigned integers can lead to unexpected behavior. |
| unsigned int x = -1; // x becomes large positive due to underflow |

| **Compliant Code** |
| --- |
| Use correct signedness and size for integers to avoid underflow or overflow. |
| int x = -1; // Proper use of signed integer |

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

| **Principles(s):**  1. Validate Input Data: Ensures data types match expected inputs, preventing malicious or erroneous data from causing issues like underflows  4. Keep It Simple: Using correct data types simplifies code logic and reduces the risk of type-related errors  10. Adopt a Secure Coding Standard: Aligns with SEI CERT rules like INT32-CPP, ensuring type safety |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | misc-non-private-member-variables-in-classes | Detects incorrect data type usage in class members. |
| Cppcheck | 2.9 | invalidFunctionArg | Identifies invalid type arguments in function calls. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard: Validate all input values** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensuring data values are within expected ranges reduces the risk of logic errors and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Failing to validate input range could lead to buffer overflow. |
| char buf[10]; gets(buf); |

| **Compliant Code** |
| --- |
| Check the length before copying data to buffer. |
| char buf[10]; fgets(buf, sizeof(buf), stdin); |

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

| **Principles(s):**  1. Validate Input Data: Validates input values to prevent buffer overflows or out-of-range errors  7. Sanitize Data Sent to Other Systems: Ensures inputs are safe before processing or passing to other components  8. Practice Defense in Depth: Input validation is a critical layer in preventing exploits |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S5986 | Detects unvalidated input vulnerabilities |
| Clang Static Analyzer | 15.0 | security.insecureAPI.gets | Flags use of unsafe functions like gets() |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard: Avoid buffer overflows with strings** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Improper string handling is a leading cause of security issues such as overflows. |

| **Noncompliant Code** |
| --- |
| Using strcpy without bounds checking leads to overflow. |
| char dest[5]; strcpy(dest, "toolong"); |

| **Compliant Code** |
| --- |
| Use strncpy with buffer size. |
| char dest[5]; strncpy(dest, "ok", sizeof(dest)); |

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

| **Principles(s):**  1. Validate Input Data: Ensures string inputs do not exceed buffer sizes  4. Keep It Simple: Using safe string functions reduces complexity in string handling  10. Adopt a Secure Coding Standard: Follows SEI CERT STR07-CPP for safe string operations |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | bufferAccessOutOfBounds | Detects buffer overflow risks in string operations |
| Clang-Tidy | 15.0 | security.insecureAPI.strcpy | Flags unsafe string functions like strcpy() |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard: Use parameterized queries** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL injection occurs when untrusted input is concatenated into SQL statements. |

| **Noncompliant Code** |
| --- |
| Concatenating user input into SQL query is unsafe. |
| std::string query = "SELECT \* FROM users WHERE name = '" + input + "'"; |

| **Compliant Code** |
| --- |
| Use prepared statements to avoid injection. |
| // Example with library specific method stmt->setString(1, input); |

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

| **Principles(s):**  1. Validate Input Data: Parameterized queries prevent malicious input from altering SQL logic  7. Sanitize Data Sent to Other Systems: Ensures safe data interaction with databases  8. Practice Defense in Depth: Adds a layer of protection against injection attacks |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S2077 | Detects SQL injection vulnerabilities in string concatenation |
| FindBugs | 3.0.1 | SQL\_INJECTION | Identifies unsafe SQL query construction |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard: Prevent memory leaks** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Memory leaks lead to exhaustion and can cause denial-of-service. |

| **Noncompliant Code** |
| --- |
| Memory is allocated but never deallocated. |
| int\* ptr = new int[10]; // no delete[] called |

| **Compliant Code** |
| --- |
| Use smart pointers or ensure delete[] is called. |
| 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):**  4. Keep It Simple: Smart pointers simplify memory management, reducing leak risks  9. Use Effective Quality Assurance Techniques: Static analysis can detect potential leaks  10. Adopt a Secure Coding Standard: Aligns with SEI CERT MEM52-CPP for resource management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18 | Memcheck | Detects memory leaks and invalid memory accesses |
| Clang Static Analyzer | 15.0 | unix.Malloc | Flags unreleased memory allocations |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard: Do not use assert for runtime logic** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions may be disabled in production, making them unreliable for enforcing security. |

| **Noncompliant Code** |
| --- |
| Using assert to validate user input. |
| assert(user\_input != NULL); |

| **Compliant Code** |
| --- |
| Use explicit runtime checks. |
| if (user\_input == NULL) { throw std::invalid\_argument("null input"); } |

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

| **Principles(s):**  1. Validate Input Data: Runtime checks ensure robust input validation  8. Practice Defense in Depth: Persistent checks add reliability beyond assertions  10. Adopt a Secure Coding Standard: Follows SEI CERT ERR09-CPP for error handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-err09-cpp | Detects misuse of assertions for runtime checks |
| Cppcheck | 2.9 | assertWithSideEffect | Flags assertions with side effects |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard: Catch and handle exceptions safely** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Failing to catch exceptions can lead to crashes or exposure of sensitive data. |

| **Noncompliant Code** |
| --- |
| Application crashes on unexpected input. |
| int value = std::stoi(user\_input); |

| **Compliant Code** |
| --- |
| Use try-catch block to handle conversion errors. |
| try { int value = std::stoi(user\_input); } catch (...) { /\* handle error \*/ } |

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

| **Principles(s):**  1. Validate Input Data: Exception handling ensures robust input processing  3. Architect and Design for Security: Proper exception handling is part of resilient design  10. Adopt a Secure Coding Standard: Aligns with SEI CERT ERR-CPP for exception safety |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cert-err61-cpp | Detects unhandled exceptions in critical paths |
| SonarQube | 9.9 | cpp:S1003 | Ensures proper exception handling practices |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard: Avoid data races and ensure thread safety** |
| --- | --- | --- |
| Concurrency | STD-008-CPP | Concurrent access to shared resources without proper synchronization can lead to unpredictable behavior and security flaws. |

| **Noncompliant Code** |
| --- |
| Multiple threads writing to a shared variable without synchronization. |
| int counter = 0; void increment() { ++counter; } // Not thread safe |

| **Compliant Code** |
| --- |
| Use mutex to protect access to shared resources. |
| #include <mutex> int counter = 0; std::mutex m; void increment() {  std::lock\_guard<std::mutex> lock(m);  ++counter; } |

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

| **Principles(s):**  3. Architect and Design for Security: Thread safety is a design consideration in concurrent systems  8. Practice Defense in Depth: Synchronization adds robustness against concurrency issues.  10. Adopt a Secure Coding Standard: Aligns with SEI CERT CON43-CPP for thread safety |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | clang-15.0 | tsan | Detects data races in multithreaded applications |
| Cppcheck | 2.9 | raceCondition | Identifies potential concurrency issues |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard: Validate and sanitize file paths** |
| --- | --- | --- |
| File Handling | STD-009-CPP | Improper file path handling can lead to path traversal attacks, exposing sensitive files. |

| **Noncompliant Code** |
| --- |
| User input is used directly to open a file. |
| std::ifstream file(user\_input); |

| **Compliant Code** |
| --- |
| Sanitize and validate file paths before using them. |
| // Check against whitelist or ensure it's within allowed directory if (is\_valid(user\_input)) {  std::ifstream file(user\_input); } |

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

| **Principles(s):**  1. Validate Input Data: Ensures file paths are safe and within expected directories  7. Sanitize Data Sent to Other Systems: Sanitizes file paths to prevent unauthorized access  8. Practice Defense in Depth: Adds validation to file access controls |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S5693 | Detects unsafe file path handling |
| Clang-Tidy | 15.0 | cert-fio42-c | Flags potential path traversal vulnerabilities |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard: Always release acquired resources** |
| --- | --- | --- |
| Resource Cleanup | STD-010-CPP | Failing to release resources like memory, file handles, or locks can cause resource exhaustion or deadlocks. |

| **Noncompliant Code** |
| --- |
| Opened file is never closed, leading to file descriptor leaks. |
| FILE\* fp = fopen("log.txt", "w"); // no fclose(fp); |

| **Compliant Code** |
| --- |
| Use RAII or ensure proper resource release. |
| #include <fstream> void log() {  std::ofstream log\_file("log.txt");  log\_file << "entry"; } // closed automatically |

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

| **Principles(s):**  4. Keep It Simple: RAII simplifies resource management, reducing leak risks  9. Use Effective Quality Assurance Techniques: Static analysis detects resource leaks  10. Adopt a Secure Coding Standard: Aligns with SEI CERT MEM51-CPP for resource cleanup |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0 | unix.API | Detects unreleased resources like file handles |
| Valgrind | 3.18 | Memcheck | Identifies resource leaks and lock issue Identifies resource leaks and lock issues |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation is strategically integrated into Green Pace’s DevSecOps lifecycle to enforce compliance with the defined C/C++ coding standards, aligning with the collaborative roles of Development (DEV), Security (SEC), and Operations (OPS) across the Pre-production and Production phases.

In the Pre-production Phase, automation is embedded as follows:

* Assess and Plan: Security tools like SonarQube (v9.9) are configured to analyze the threat landscape and prioritize coding standards (e.g., STD-004-CPP for SQL injection) in the backlog, ensuring regulatory compliance is addressed early.
* Design: Static analysis tools such as Clang-Tidy (v15.0) enforce OWASP best practices by detecting design-level issues, such as improper data type usage (STD-001-CPP), during code reviews in CI pipelines (e.g., Jenkins).
* Build: Secure builds are enforced using trusted repositories with tools like Cppcheck (v2.9) to identify vulnerabilities like buffer overflows (STD-003-CPP) or memory leaks (STD-005-CPP). Build failures are triggered for non-compliant code.
* Verify and Test: Vulnerability scanning with SonarQube and Clang Static Analyzer (v15.0) conducts security testing for issues like unvalidated inputs (STD-002-CPP) or unsafe file handling (STD-009-CPP). ThreadSanitizer (clang-15.0) is used for concurrency testing (STD-008-CPP). Digitally signed sources ensure integrity.

In the Production Phase, automation continues to ensure compliance:

* Transition and Health Check: During deployment, Valgrind (v3.18) verifies resource cleanup (STD-010-CPP), and penetration testing tools validate security settings. Configurations are checked against compliance requirements.
* Monitor and Detect: SIEM systems (e.g., Splunk) collect logs from tools like SonarQube and Clang-Tidy, enabling real-time detection of non-compliance (e.g., assertion misuse in STD-006-CPP). Event alerting flags anomalies.
* Respond: Automated scripts block non-compliant deployments, and rollback mechanisms are triggered if vulnerabilities like exception mishandling (STD-007-CPP) are detected.
* Maintain and Stabilize: Systems are compared against security baselines using automated audits, ensuring standards are maintained post-deployment.

This automation leverages the interconnected DEV, SEC, and OPS roles, embedding tools like SonarQube, Clang-Tidy, Cppcheck, Valgrind, and ThreadSanitizer into the CI/CD pipeline (e.g., GitLab CI/CD). Results are reported via dashboards, fostering collaboration and transparency. By aligning with the DevSecOps lifecycle’s iterative approach, Green Pace ensures continuous enforcement of coding standards, reducing vulnerabilities and maintaining compliance throughout development and deployment.

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

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Data stored in databases, files, or storage devices must be encrypted using AES-256 to protect against unauthorized access or theft. This policy applies to all sensitive data, including customer PII and proprietary data, stored on Green Pace servers or third-party cloud platforms. Encryption at rest ensures that even if physical or logical access is gained, the data remains unreadable without decryption keys, which are managed securely via a key management system (KMS). This is critical for compliance with data protection regulations and to mitigate breach risks. |
| Encryption in flight | Data transmitted over networks, including internal APIs and external communications, must use TLS 1.3 to encrypt data in transit. This applies to all network traffic containing sensitive information, such as user credentials or financial data, to prevent man-in-the-middle attacks. The policy ensures confidentiality and integrity during transmission and is enforced by configuring servers and clients to reject unencrypted connections. Regular audits verify TLS configuration compliance. |
| Encryption in use | Data processed in memory should be protected where feasible, using techniques like secure enclaves (e.g., Intel SGX) or homomorphic encryption for sensitive computations. This policy applies to applications handling cryptographic keys or highly sensitive data during runtime, such as financial transactions. It aims to protect data from memory-based attacks like buffer dumps and is implemented where supported by hardware and software capabilities, enhancing defense-in-depth. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users and systems must authenticate using multi-factor authentication (MFA) before accessing Green Pace systems or applications. This includes user logins and API interactions, verified through identity providers like Okta. The policy applies to all access points to ensure only verified identities can interact with systems, preventing unauthorized access. New users are provisioned with MFA during onboarding, and authentication logs are audited to detect anomalies. |
| Authorization | Access to resources, including files and databases, is restricted based on user roles and the principle of least privilege, enforced via role-based access control (RBAC). User access levels are defined during onboarding or role changes, and access to sensitive data (e.g., customer records) requires explicit approval. This policy applies to all systems to limit exposure and is audited to track access attempts, ensuring compliance and security. |
| Accounting | All user actions, including logins, database changes, new user additions, and file accesses, are logged and audited in real-time using a centralized SIEM (Security Information and Event Management, e.g., Splunk). Logs include timestamps, user IDs, and action details for traceability. This policy applies to all system interactions to enable forensic analysis and compliance reporting, ensuring accountability and detecting unauthorized or suspicious activities. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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