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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Ensuring that all input data is validated by checking for type, length, format before it is passed. Improper input handling may lead to buffer overflows and injections. |
| 1. Heed Compiler Warnings | Treat all compile warning as errors, address these proactively to identify and eliminate potential vulnerabilities during development. |
| 1. Architect and Design for Security Policies | Security should be embedded throughout all development processes, from design phase to delivery of product. Security should not be an afterthought and must be treated as a fundamental part of the software life cycle. |
| 1. Keep It Simple | Keep designs simple. Complex designs can create vulnerability and avoid overly complicated logic or unnecessary features. |
| 1. Default Deny | Access should be denied by default, unless explicitly allowed. Avoids accidental exposure. |
| 1. Adhere to the Principle of Least Privilege | Users operate to minimum access, only necessary level access. Reduces risk of accidental or malicious use, limiting the impacts of components. |
| 1. Sanitize Data Sent to Other Systems | Ensure data outputs to other systems are sanitized to prevent cross-system vulnerabilities. |
| 1. Practice Defense in Depth | Implement layers of security for redundancy in the events that one fails. Firewalls, encryption, validation input and access control prevent multilayer protection on critical assets. |
| 1. Use Effective Quality Assurance Techniques | Security should be part of the QA process. Static and dynamic testing, code reviews and peer reviews focused on security flaws should be always implemented. Automation can complement these efforts. |
| 1. Adopt a Secure Coding Standard | Use established secure code standards, such as SEI CERT C++ guidelines during development process. |

### 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] | INT36-C: Converting a pointer to integer or integer to pointer |

| **Noncompliant Code** |
| --- |
| This code casts a pointer to an integer, which may result in truncation or invalid values on 64-bit systems |
| **void** f(**void**) {  **char** \*ptr;    /\* ... \*/    unsigned **int** number = (unsigned **int**)ptr;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Any valid pointer to void can be converted to intptr\_t or untptr\_t and back with no change in value. C standard guarantees that a pointer to void may be converted to or from a pointer to any object type and back again. The result must compare equal to the original pointer |
| #include <stdint.h>    **void** f(**void**) {  **char** \*ptr;    /\* ... \*/  **uintptr\_t** number = (**uintptr\_t**)ptr;    /\* ... \*/  } |

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

| **Principles(s):**   1. **Principle 1: Validate Input –** Ensure data conversions are valid, predictable. Without validation, casting pointers can lead to unsafe or undefined behaviors. 2. **Principle 9: Fail Securely –** Unsafe casts can cause runtime errors, expose memory locations. Ensuring proper casting means programs will fail in a controlled and predictable way. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | clang-analyzer-core.CastToVoid | Warns about unsafe casts involving pointers |
| Coverity | 2023.3 | UNSAFE\_CAST | Detects unsafe type conversions |
| CodeSonar | 7.4 | PointerCastViolation | Flags invalid or non-portable point caster |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | INT32-C: Ensure that integer conversions do not result in loss or misinterpreted data |

| **Noncompliant Code** |
| --- |
| This code converts a long value to an int, which can result in data loss if the value exceeds the range of int |
| void f(long l) {  int i = (int)l; // Possible truncation or data loss  } |

| **Compliant Code** |
| --- |
| This version check if the conversion is safe before proceeding. |
| #include <limits.h>  #include <stdio.h>  void f(long l) {  if (l > INT\_MAX || l < INT\_MIN) {  printf("Out-of-range value\n");  return;  }  int i = (int)l;  } |

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

| **Principles(s):**   1. **Principle 1: Validate Input** – Preventing overflow ensures numeric inputs are within safe limits. 2. **Principle 9: Fail Securely** – Failing without validating signed arithmetic can lead to exploitable crashes or logic errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S3510 | Detects unsafe arithmetic operations |
| Clang-Tidy | 17.0 | Cert-int32-c | Warns about possible overflow conditions |
| Coverity | 2023.3 | INTEGER\_OVERFLOW | Finds overflow risks in arithmetic |
|  |  |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | STR30-C: Do not attempt to modify string literals |

| **Noncompliant Code** |
| --- |
| Modifying a string literal results in undefined behaviors. |
| void f(void) {  char \*str = "Hello";  str[0] = 'J'; // Undefined behavior  } |

| **Compliant Code** |
| --- |
| Use a character array if you need a modifiable string |
| void f(void) {  char str[] = "Hello";  str[0] = 'J'; // Safe  } |

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

| **Principles(s):**   1. **Principle 1: Validate Input** – Prevents buffer overflows and string truncation. 2. **Principle 10: Defense in Depth** – Adds a layer of protection during string operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S1751 | Detects unsafe string usage |
| Clang-Tidy | 17.0 | Cert-str31-c | Enforces safe string manipulation |
| Fortify | 23.2 | STRING\_MANIPULATION | Flags unsafe strcpt/strcat |
|  |  |  |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | IDS51-J: Properly encode or escape output |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries allows for injection |
| void query(std::string userInput) {  std::string sql = "SELECT \* FROM users WHERE name = '" + userInput + "'";  // execute sql query...  } |

| **Compliant Code** |
| --- |
| Use parameterized queries or sanitization functions to escape input |
| void query(std::string userInput) {  // Use a proper database API with parameterized queries (pseudo-code)  db.prepare("SELECT \* FROM users WHERE name = ?");  db.bind(userInput);  db.execute();  } |

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

| **Principles(s):**   1. **Principle 6: Keep It Simple** – Ensures predictable behavior by requiring proper initialization. 2. **Principle 8: Secure the Weakest Link** – Targets common low-level vulnerabilities that can lead to severe exploits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S1481 | Flags use of uninitialized variables |
| Clang-Tidy | 17.0 | Cert-exp33-c | Detects uninitialized variable use |
| Valgrind | 3.21 | Memcheck | Runtime tool for memory tracking |
|  |  |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CCP] | MEM50-CPP: Do not access free memory |

| **Noncompliant Code** |
| --- |
| Accessing memory after being freed causes undefined behavior |
| void f(void) {  int \*p = new int(42);  delete p;  \*p = 13; // Use after free  } |

| **Compliant Code** |
| --- |
| Set pointers to nullptr after freeing to avoid accidental access |
| void f(void) {  int \*p = new int(42);  delete p;  p = nullptr; // Safe, now cannot dereference  } |

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

| **Principles(s):**   1. **Principle 5: Least Privilege** – Prevents access to memory that no longer belongs to a process. 2. **Principle 9: Fail Securely** – Ensures the system doesn't crash or expose sensitive data due to dangling pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S2259 | Detects use-after-all-free or null dereference |
| Clang-Tidy | 17.0 | Clang-analyzer-cplusplus.NewDelete | Identifies use of deleted memory |
| Valgrind | 3.21 | Memcheck | Runtime use-after-free detection |
|  |  |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | MSC30-C: Do not use assertations to validate function arguments |

| **Noncompliant Code** |
| --- |
| Assertions can be disabled in production builds, removing critical validation |
| #include <assert.h>  void f(int index) {  assert(index >= 0);  // ...  } |

| **Compliant Code** |
| --- |
| Use explicit runtime checks instead of assertions for validation |
| #include <stdio.h>  void f(int index) {  if (index < 0) {  printf("Invalid argument\n");  return;  }  // ...  } |

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

| **Principles(s):**   1. **Principle 6: Keep It Simple** – Removes unnecessary complexity from codebases. 2. **Principle 8: Secure the Weakest Link** – Eliminates redundant logic that could hide vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S1172 | Flags unused parameters and dead code |
| Clang-Tidy | 17.0 | Misc-unused-using-decls | Detects dead code and unreachable logic |
| Cppcheck | 2.12 | unusedFunction | Identifies unused or unreachable functions |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | ERR55-CPP: Honor exception specifications |

| **Noncompliant Code** |
| --- |
| Throwing exceptions from a function not marked noexcept violates contract. |
| void f() noexcept {  throw std::runtime\_error("Error"); // Undefined behavior  } |

| **Compliant Code** |
| --- |
| Only throw from functions that do not declare noexcept |
| void f() {  throw std::runtime\_error("Error"); // Safe  } |

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

| **Principles(s):**   1. **Principle 9: Fail Securely** – Ensures graceful failure handling rather than abrupt terminations. 2. **Principle 10: Defense in Depth** – Allows layered error recovery mechanisms to function correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S3512 | Detects calls to exit(), abort(), ect |
| Clang-Tidy | 17.0 | Cert-err50-cpp | Flags premature process termination |
| Coverity | 2023.3 | UNREACHABLE | Detect dead code paths after abort |
|  |  |  |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | EXP34-C: Do not dereference null pointers |

| **Noncompliant Code** |
| --- |
| Dereferencing a null pointer causes undefined behavior |
| void f(int \*ptr) {  \*ptr = 42; // If ptr is null, this is a crash  } |

| **Compliant Code** |
| --- |
| Check for null before dereferencing |
| void f(int \*ptr) {  if (ptr != nullptr) {  \*ptr = 42;  }  } |

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

| **Principles(s):**   1. **Principle 1: Validate Input** – Ensures that all pointers are checked for null before being used, reducing chances of crashes. 2. **Principle 9: Fail Securely** – Prevents undefined behavior that can occur when dereferencing null, which could lead to exploitable conditions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S2259 | Detects potential null pointer dereference |
| Clang-Tidy | 17.0 | Clang-analyzer-core.NullDereference | Static check for null dereference at compile time |
| Coverity | 2023.3 | NULL\_RETURNS | Finds null checks that are missing before dereference |
| Fortify | 23.2 | NULL\_POINTER | Identifies paths that lead to null pointer use |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | STR32-C: Do not pass a non-null pointer to a function that expects a null pointer |

| **Noncompliant Code** |
| --- |
| Passing an uninitialized pointer into fopen() can result in unexpected behavior |
| char \*filename;  FILE \*fp = fopen(filename, "r"); // filename not initialized |

| **Compliant Code** |
| --- |
| Always initialize and validate pointers before use |
| char \*filename = "file.txt";  FILE \*fp = fopen(filename, "r"); // Valid usage |

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

| **Principles(s):**   1. **Principle 1: Validate Input** – Ensures that pointers passed into functions are initialized and valid for the expected behavior. 2. **Principle 6: Keep It Simple** – Reduces hidden logic errors that result from misuse of pointer intent. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S6026 | Detects unsafe parameter handling in pointer API’s |
| Clang-Tidy | 17.0 | Cert-str32-c | Warns when inappropriate non-null values are passed |
| Coverity | 2023.3 | CHECK\_RETURN, MISSING\_CHECK | Flags failure to validate pointer usage exceptions |
|  |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | FIO30-C: Exclude user input from format strings |

| **Noncompliant Code** |
| --- |
| Using user input as the format string may allow format string attacks |
| char input[100];  scanf("%s", input);  printf(input); // Dangerous |

| **Compliant Code** |
| --- |
| Always used fixed format strings |
| char input[100];  scanf("%s", input);  printf("%s", input); // Safe |

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

| **Principles(s):**   1. **Principle 1: Validate Input** – Ensures format strings are never influenced by untrusted user input. 2. **Principle 10: Defense in Depth** – Prevents format string vulnerabilities by enforcing input/output separation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | S5693 | Detects unsanitized user input used in format strings |
| Clang-Tidy | 17.0 | Cert-fio30-c | Detects format string injection vulnerabilities |
| Coverity | 2023.3 | TAINTED\_SCALAR | Flags tainted reaching format function parameters |
|  |  |  |  |

### 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 ensure that Green Pace remains compliant with coding standards, security tools should be directly integrated into all pipelines and across every stage of the development process. During the Verify phase, static analysis tools such as SonarQube, Clang-Tidy, Coverity, and Fortify should be used to detect violations of SEI CERT rules. These tools provide real-time feedback, identifying vulnerabilities such as buffer overflows and format string injections before the code reaches production.**

**Tool rules from SonarSource, such as S5144 and S2259, map directly to CERT standards and align with widely recognized secure coding practices. During the Plan and Create phases, IDE-integrated plugins can preemptively flag potential issues as code is written. Post-analysis findings should be triaged and prioritized based on the severity and impact of the vulnerabilities.**

**By embedding these tools into the CI/CD workflow—before, during, and after builds—Green Pace can minimize vulnerabilities, reduce human error, and uphold a strong defense-in-depth (DiD) security posture throughout the software lifecycle.**

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | Critical | 1 |
| STD-003-CPP | High | Likely | Low | High | 2 |
| STD-004-CPP | Medium | Likely | Low | Medium | 3 |
| STD-005-CPP | High | Likely | Medium | Critical | 1 |
| STD-006-CPP | Low | Likely | Low | Low | 4 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 3 |
| STD-008-CPP | High | Likely | Low | High | 2 |
| STD-009-CPP | Medium | Possible | Low | Medium | 3 |
| STD-010-CPP | High | Likely | Low | High | 1 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to securing data stored on a physical or virtual device. All sensitive data should be encrypted using AES-256 or stronger. This policy applies to all physical or cloud-based storage to prevent data exposure incase of unauthorized access is obtained. Encryption keys will remain secure with a Key Management System. |
| Encryption in flight | Encryption in flight is data that travels between systems, users and applications. To prevent man-in-the-middle (MITM) attacks, TLS 1.2 and higher will be utilized for all internal and external communication. This includes all API, portals, remote development and file transfers. |
| Encryption in use | Encryption in use protects data being processed. Confidential operations that involve Personally Identifiable Information (PII) will be isolated to trusted environments only. This applies to all workflows, where data is exposed during runtime, which will mitigate memory scraping or malware intercepts. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures users are who the say they are. Green Pace will enforce multi-factoring authentication (MFA) across all systems, portals and development tools. All user logins must be unique, logging enabled and tied to roles to prevent unauthorized access to systems they should not be accessing. Passwords will follow strict complex standards and rotate on a regular basis, typically every 6 months a force password change will be implemented. |
| Authorization | Authorization governs what users are allowed to access and modify. Role based access will be utilized with the principle of least privilege. Meaning access will only be authorized for the individual into certain aspects and isolating other systems to prevent unauthorized use. The users and roles will be reviewed quarterly and purged out of any unnecessary data. Any changes to roles must be logged and approved by security personnel. This policy is critical for control of sensitive data. |
| Accounting | Accounting tracks and logs all user interactions. These logs will include timestamps, changes to database, new user creation, permission changes and files accessed. Green Pace uses centralized logging and monitoring tools to record and alert of any suspicious behaviors, such as a user with non-authorized access into certain systems. These logs will be retained for 12 months and reviewed during audits. This policy supports regulatory compliance, audit readiness and able to respond to incidents. |

**\***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 |  |
| 2.0 | 04/18/2024 | Complete full security policy per SEI CERT and security standards | Alexander DeMarco | SNHU – Professor O.B. |
|  |  |  |  |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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