**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 input data cannot exceed the allocated bounds of expected input by utilizing std::string. Ensure the data is valid syntactically for input scheme as well as semantically for the context for which it applies. (e.g. An SSN is ###-##-####, and should not be entered in an email field). |
| 1. Heed Compiler Warnings | After ensuring that the compiler settings are appropriately configured to detect anomalous design and unsafe practices, utilizing the warning messages is required for code mergers to be accepted. Code that produces compiler warnings will not be considered. |
| 1. Architect and Design for Security Policies | The foremost design consideration that must be considered. Designing to support security policies ensures that code execution does not elevate a user’s effective access to the system under any circumstances and prevents unauthorized execution of code. |
| 1. Keep It Simple | The simplest solution is often the most straightforward one. To err on the side of succinct code that provides the base functionality required without anticipating edge cases is sufficient unless tests and review prove otherwise. |
| 1. Default Deny | When designing validation methods, always design for the most rapid failure possible, and if any ambiguity exists, return an indication of failure rather than success. This ensures that in the case a validation bug exists, the outcome is that the system will deny. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege exists to prevent unauthorized users from accessing or changing data when they should not have the right. This prevents external attacks from acquiring privilege that allows for destructive behavior, as well as preventing internal users from accidental, non-malicious destruction when they should be unable to do so. |
| 1. Sanitize Data Sent to Other Systems | Data sent to other systems must not contain internal data points, private information, or other sensitive data either about user data or the program’s execution. |
| 1. Practice Defense in Depth | Layered security is the only valid and accepted way to ensure security on a system. Accounting for failures with other software or hardware solutions to security that apply to technologies being developed will allow the designer to prevent unauthorized access to systems executing said software. |
| 1. Use Effective Quality Assurance Techniques | Test cases and assertions should be used as part of the developer’s toolkit when creating their assigned solutions. The final product will not use assertions and test cases during execution, however, so validation is of the utmost importance, as the debug assertions will not be utilized in any form. |
| 1. Adopt a Secure Coding Standard | Following the above principles will ensure that security-minded development is carried out in most applications, but it is still the requirement of each developer to review not only this document but the [SEI CERT C++ Coding Standard](https://wiki.sei.cmu.edu/confluence/) webpage to stay informed of the most up-to-date practices. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Test data types before performing type-specific operations |

| **Noncompliant Code** |
| --- |
| Attempting to execute code from a user input expecting an integer can produce unexpected results: |
| int x;  cin >> x;  cout << “the square of “ << x << “ is “ << (x\*x) << endl; |

| **Compliant Code** |
| --- |
| Properly gathering user input and performing verification provides more stable execution: |
| string user\_input;  cout << “Type a number : ”;  cin >> user\_input;  int x = atoi(user\_input.c\_str());  cout << “the square of “ << x << “ is “ << (x\*x) << endl; |

| **Fail-Safe defaults / Complete Mediation:** should always continue execution even if an error is encountered |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| StyleCop | For Visual Studio 2019 | StyleCop is configurable | Configure tool to detect lacking type checking |

#### 

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Expecting data values without testing can lead to unexpected results |

| **Noncompliant Code** |
| --- |
| Utilizing data values with an expected value can produce unexpected results: |
| int x;  cout << "Type a number : ";  cin >> x;  for (int i = 0; i < x; i++) { /\*do something\*/ } |

| **Compliant Code** |
| --- |
| Correctly gathering input and testing for acceptable values provides more stable execution: |
| string user\_input;  cout << "Type a number : ";  cin >> user\_input;  int x = atoi(user\_input.c\_str());  for (int i = 0; i < x; i++) { /\*do something\*/ } |

| **Fail-Safe defaults / Complete Mediation:** should always continue execution even if an error is encountered |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| StyleCop | For Visual Studio 2019 | StyleCop is configurable | Configure tool to detect lacking type checking |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | User input must always be gathered by a std::string |

| **Noncompliant Code** |
| --- |
| Gathering user input into a character array can cause overflow and allow private code execution: |
| char user\_input[20];  cout << "Enter a value: ";  cin >> user\_input;  cout << "You entered: " << user\_input << endl; |

| **Compliant Code** |
| --- |
| Gathering user input into a string array is boundless and less error prone: |
| string user\_input;  cout << "Enter a value: ";  cin >> user\_input;  cout << "You entered: " << user\_input << endl; |

| **Economy of Mechanism / Least common mechanism –** always gathering user input in the same manner reduces differences across the application and reduces total testing required. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | For Visual Studio 2019 | <https://rules.sonarsource.com/cpp/type/Vulnerability/RSPEC-1081> | Insecure Functions should not be used: preventing buffer overflow |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL injection by sanitization of user inputs |

| **Noncompliant Code** |
| --- |
| Sending a SQL query without checking for potentially dangerous SQL strings can cause unauthorized access: |
| string user\_input;  cout << "Enter a query: ";  cin >> user\_input;  SQLquery(user\_input); |

| **Compliant Code** |
| --- |
| Sanitizing and utilizing a try / catch to prevent bad queries is a safer practice: |
| [… Same as previous code for user input …]  for (size\_t i = 0; i < user\_input.length(); i++) {  switch (user\_input[i]) {  case '<':case '>':case '=':case '\'':case '\"':case '\\':case '&':case '|':case '{':case '}':  throw(SQLException("BAD SQL DETECTED"));  default:  break;  }  }  SQLquery(user\_input); |

| **Complete Mediation –** checking user input for malicious code and breaking characters is a best practice |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| StyleCop | For Visual Studio 2019 | StyleCop is configurable | Configure to detect missing sanitization of string data |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Prevent access to memory beyond bounds of allocation. |

| **Noncompliant Code** |
| --- |
| Iteration of an array past its bounds can access external memory: |
| char word[20] = "Test Words";  for (size\_t i = 0; i < 30; i++)  {  cout << word[i];  } |

| **Compliant Code** |
| --- |
| Testing the bounds of an array is required to prevent external memory access: |
| char word[20] = "Test Words";  for (size\_t i = 0; i < sizeof(word); i++)  {  cout << word[i];  } |

| **Complete mediation –** preventing out-of-bounds and other undefined behaviors is best practice. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | For Visual Studio 2019 | <https://rules.sonarsource.com/cpp/type/Vulnerability/RSPEC-1081> | Insecure Functions should not be used: preventing buffer overflow |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions should be used for proper debug testing, but code must work with assertions omitted. |

| **Noncompliant Code** |
| --- |
| Using an assertion to test a value for proper execution will not function as expected if compiled with no asserts: |
| int x;  cin >> x;  assert(x != 0);  cout << 2 / x << endl; |

| **Compliant Code** |
| --- |
| Using an assertion to verify an expected value for debugging purposes that has no bearing on final code execution: |
| int x;  cin >> x;  if (x != 0) {  float y = 2.0f / x;  assert(y != 0);  cout << y << endl;  } |

| **Fail-safe defaults / Open Design –** requiring an assert in production code obfuscates the goal of the code and will lead to issues when deploying with no debug-level code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| StyleCop | For Visual Studio 2019 | StyleCop is configurable | Configure to Error on methods or assignment inside asserts |

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions must be thrown when code execution cannot continue. |

| **Noncompliant Code** |
| --- |
| Verifying SQL injection attempt but resuming execution may cause issues: |
| for (size\_t i = 0; i < user\_input.length(); i++)  {  switch (user\_input[i])  {  case '<':case '>':case '=':case '\'':case '\"':case '\\':case '&':case '|':case '{':case '}':  break;  default:  break;  }  }  SQLquery(user\_input); |

| **Compliant Code** |
| --- |
| Verifying SQL injection attempt and throwing a custom exception provides data on error: |
| for (size\_t i = 0; i < user\_input.length(); i++)  {  switch (user\_input[i])  {  case '<':case '>':case '=':case '\'':case '\"':case '\\':case '&':case '|':case '{':case '}':  throw(SQLException("BAD SQL DETECTED"));  default:  break;  }  }  SQLquery(user\_input); |

| **Psychological acceptability –** If the program must throw an exception to notify the user of something, an error message will serve to give a human-readable reason that execution has quit. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | For Visual Studio 2019 | <https://rules.sonarsource.com/cpp/tag/error-handling/RSPEC-2486> | Exceptions should not be ignored – usage of try / catch blocks |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-008-CPP | Code that may throw exceptions must by in a try/catch block |

| **Noncompliant Code** |
| --- |
| Execution is terminated abruptly if unhandled exceptions are encountered: |
| int x;  cin >> x;  float y = 1.0f / x; |

| **Compliant Code** |
| --- |
| Execution may continue if potential exceptions are accounted for: |
| int x;  cin >> x;  try {  float y = 1.0f / x;  }  catch (const std::exception&) {} |

| **Economy of mechanism /Least common mechanism –** adhering to try/catch blocks around possible exceptions allows for developers to use a consistent style and handle caught errors consistently. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | For Visual Studio 2019 | <https://rules.sonarsource.com/cpp/tag/error-handling/RSPEC-2486> | Exceptions should not be ignored – usage of try / catch blocks |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory** | STD-009-CPP | Do not access freed memory or pointers |

| **Noncompliant Code** |
| --- |
| Accessing freed memory is unstable and can cause vulnerabilities if execution continues |
| string\* s = new string();  delete s;  cout << s->c\_str(); |

| **Compliant Code** |
| --- |
| Proper memory management implemented to allow access and free the pointer when done |
| string\* s = new string();  cout << s->c\_str();  delete s; |

| **Complete mediation –** Developers must take care to test that memory is not freed, and should not rely on any undefined behavior for execution, such as accessing freed memory pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | For Visual Studio 2019 | <https://rules.sonarsource.com/cpp/type/Bug/RSPEC-3529> | Freed memory should not be used – preventing undefined behavior |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Size** | STD-010-CPP | Ensure that numeric functions do not overflow |

| **Noncompliant Code** |
| --- |
| Unchecked addition function can cause integer overflow: |
| template <typename T>  T add\_numbers(T const& start, T const& increment, unsigned long int const& steps)  {  T result = start;  for (unsigned long int i = 0; i < steps; ++i)  {  result += increment;  }  return result;  } |

| **Compliant Code** |
| --- |
| Testing against the maximum limit for a type prevents overflow and returns an actionable value of NULL to indicate overflow would have occurred: |
| template <typename T>  T add\_numbers(T const& start, T const& increment, unsigned long int const& steps)  {  T result = start;  for (unsigned long int i = 0; i < steps; ++i)  {  if (std::numeric\_limits<T>::max() - increment < result)  return NULL;  result += increment;  }  return result;  } |

| **Fail-safe defaults / Least common mechanism –** Devs ensuring that their operations do not cause over/underflow will ensure the program executes as expected and does not unexpectedly crash. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | For Visual Studio 2019 | <https://rules.sonarsource.com/cpp/type/Bug/RSPEC-3949> | Integral operations should not overflow – undefined behavior |

### Defense-in-Depth Illustration

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





Automation will be used for the enforcement of and compliance to the standards defined in this policy. The application of automation should be utilized to perform unit testing upon code when it is built and remitted to source control. This includes branches that are not production code, to verify their efficacy and perform regression testing. Code analyzers within Visual Studio must not be tampered with, and modifications to build switches and preprocessor checks will not be allowed for code acceptance to production. All code seeking to be merged to production must undergo develops rigor testing through tens of thousands of cycles using verified input data and bad requests for proofing. Production code will always publish minimal logging for health checking, event alerts, and intrusion detection. Critical component failure requires on-duty IT notification and a sub-one-hour response time.

### Summary of Risk Assessments

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

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encrypted data not being utilized must be stored in an encrypted state that is decipherable via a PK (private key) or by utilizing one-sided verification with an AES+ level encryption algorithm. |
| Encryption at flight | Transferring data must be done with AES+ encryption with the receiving end performing decryption with a preestablished asymmetrical key. |
| Encryption in use | Once data is decrypted for use, it must be discarded or re-encrypted before transmitting or storing, immediately once the decrypted data is not needed. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Used to verify a user’s login credentials.  This should initiate the Authorization step, to ensure the user is trusted. |
| Authorization | Authorizing a user checks their access privileges:  Can they log in?  Do we know where they are logged in from and trust it?  What are they allowed to do, where in the servers can they go? |
| Accounting | All actions should be logged within reason.  User creation, User login records, User production-level changes should be accounted in order to catch vulnerable accounts, intentional damages, or unauthorized access. |

### Map the Principles

|  |  |
| --- | --- |
| STD-001-CPP | Test data types before performing type-specific operations |
| Principle 1 | Validate Input Data |
| Validation of types when unknown ensures that type-specific operations are valid and do not cause uncaught exceptions. | |

|  |  |
| --- | --- |
| STD-002-CPP | Expecting data values without testing can lead to unexpected results |
| Principle 7 | Sanitize Data Sent to Other Systems |
| Data incoming to developer code is data sent to your system that you cannot trust, sanitization is the only acceptable way to proceed. | |

|  |  |
| --- | --- |
| STD-003-CPP | User input must always be gathered by a std::string |
| Principle 4 | Keep it simple |
| Standardizing user input into a single format keeps it simple. | |

|  |  |
| --- | --- |
| STD-004-CPP | Prevent SQL injection by sanitization of user inputs |
| Principle 7 | Sanitize Data Sent to Other Systems |
| Outgoing data, especially to SQL databases, must be sanitized or it risks intrusion. | |

|  |  |
| --- | --- |
| STD-005-CPP | Prevent access to memory beyond bounds of allocation. |
| Principle 2 | Heed compiler warnings |
| Modern compilers will generally catch this error, but attention must be paid nonetheless. | |

|  |  |
| --- | --- |
| STD-006-CPP | Assertions should be used for proper debug testing, but code must work with assertions omitted. |
| Principle 9 | Use effective quality assurance techniques |
| Asserts are a powerful tool for testing and verification, but care must be taken to prevent normal code execution from relying on an assertion. | |

|  |  |
| --- | --- |
| STD-007-CPP | Exceptions must be thrown when code execution cannot continue. |
| Principle 3 | Architect and design for security policies |
| If an exception must be thrown, then let it be thrown. Take care to prevent execution failure and crashes, but notifying the user to critical errors is essential. | |

|  |  |
| --- | --- |
| STD-008-CPP | Code that may throw exceptions must by in a try/catch block |
| Principle 4 | Keep it simple |
| Uniformly trapping exceptions in try / catch blocks as a rule is a good practice, and establishing it as a standard seeks to keep order and consistency in the code. | |

|  |  |
| --- | --- |
| STD-009-CPP | Do not access freed memory or pointers |
| Principle 2 | Heed compiler warnings |
| Most modern compilers will catch access of deallocated memory or pointers, but be aware when writing and reviewing code that it is possible. | |

|  |  |
| --- | --- |
| STD-010-CPP | Ensure that numeric functions do not overflow |
| Principle 9 | Use effective quality assurance techniques |
| Testing every addition and operation is costly, but applying this standard where it may matter is vital to ensuring undefined and unexpected behavior does not occur. | |

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 09/18/2022 | C++ coding policies | Michael Cannon |  |
| 1.2 | 10/09/2022 | Risk / threat assessments, automation, principle mapping | Michael Cannon |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

**Analysis tools:**

<https://rules.sonarsource.com/cpp>