**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc200891215)

[Contents 1](#_Toc200891216)

[Overview 3](#_Toc200891217)

[Purpose 3](#_Toc200891218)

[Scope 3](#_Toc200891219)

[Module Three Milestone 3](#_Toc200891220)

[Ten Core Security Principles 3](#_Toc200891221)

[C/C++ Ten Coding Standards 4](#_Toc200891222)

[Coding Standard 1 5](#_Toc200891223)

[Coding Standard 2 7](#_Toc200891224)

[Coding Standard 3 9](#_Toc200891225)

[Coding Standard 4 10](#_Toc200891226)

[Coding Standard 5 12](#_Toc200891227)

[Coding Standard 6 14](#_Toc200891228)

[Coding Standard 7 16](#_Toc200891229)

[Coding Standard 8 18](#_Toc200891230)

[Coding Standard 9 20](#_Toc200891231)

[Coding Standard 10 21](#_Toc200891232)

[Defense-in-Depth Illustration 23](#_Toc200891233)

[Project One 23](#_Toc200891234)

[Revise the C/C++ Standards 23](#_Toc200891235)

[Risk Assessment 23](#_Toc200891236)

[Automated Detection 23](#_Toc200891237)

[Automation 23](#_Toc200891238)

[Summary of Risk Assessments 24](#_Toc200891239)

[Create Policies for Encryption and Triple A 25](#_Toc200891240)

[Map the Principles 26](#_Toc200891241)

[Audit Controls and Management 29](#_Toc200891242)

[Enforcement 29](#_Toc200891243)

[Exceptions Process 29](#_Toc200891244)

[Distribution 30](#_Toc200891245)

[Policy Change Control 30](#_Toc200891246)

[Policy Version History 30](#_Toc200891247)

[Appendix A Lookups 30](#_Toc200891248)

[Approved C/C++ Language Acronyms 30](#_Toc200891249)

## 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 | Input data for any operation must be validated against what is expected prior to any operation on that data. This is essential in preventing undefined behavior, both malicious and incidental, and ensures predictable results. This also provides a means of catching potential errors upstream of any function call or incorrect specifications meant for a given function. |
| 1. Heed Compiler Warnings | Compiler warnings are not suggestions. These warnings indicate a fundamental issue within a codebase and cannot be ignored without adjudication. From an operational perspective, warnings could be indicative of potential edge cases that will result in undefined behavior and from a security perspective, this could indicate that there is a vulnerable surface area in the code base. Proper notice and adjudication are required so that potential risks are understood, categorized, and mitigated or accepted. |
| 1. Architect and Design for Security Policies | Security policies are most effective when incorporated into the design and architecture of a software package. This means that the design itself contributes to the security posture of a package when combined with security best practices within the code itself. The summation of these practices organized coherently as part of the architecture ensures the security robustness of the system. |
| 1. Keep It Simple | Long, convoluted blocks of code are more difficult to read and understand and often preform more operations than are necessary. This also has the potential to add vulnerable surface area to a code base that is unnecessary. If such a convoluted block of code is necessary to perform an operation, that is indicative of that block doing too much and that other potential solutions should be considered. |
| 1. Default Deny | If data, connections, or permissions are not explicitly expected or needed, then they are explicitly not required. No data, connections, or permissions should be allowed into or onto a system that have not been designed into the system and the necessity verified. This improves the security posture as well as improves operational efficiency. |
| 1. Adhere to the Principle of Least Privilege | Least Privilege is the defined as a process, user, or code having the least number of permissions to accomplish a task. In code, this amounts to ensuring that access to a Classes’ methods, functions, and variables are limited to the least possible scope while still maintaining functionality. Classes should not be able to access other Classes’ variables and internal functions and if data is required, that they are accessed through special methods such as “setters” and “getters”. |
| 1. Sanitize Data Sent to Other Systems | Data sent to another system should appear as if coming from a “black box”. Only the information required by the other system should be passed and this information must have no indication of what processing or transformation has occurred to it prior to its receipt. This addresses potential privacy concerns and also prevents potential reconnaissance of the system or its operation. |
| 1. Practice Defense in Depth | Defense in Depth can ensure security of a system by preventing a malicious operation via different mechanisms e.g. if a class object needs to set a variable of another class object, it would access a “setter” function instead of accessing the variable directly, and that “setter” function would validate the potential input prior to changing the variable value. This ensures that the security of the internal class object variable remains intact despite potential malicious attempts at manipulating it either directly or by accessing another class object. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance processes validate that a system meet the requirements set out for that system. This includes security posture. The quality of a code base is only as valid as the techniques used to validate it. Effective QA practices are the only way to ensure that a system and codebase are in fact living up to the requirements for it. |
| 1. Adopt a Secure Coding Standard | A Secure Coding Standard provides a number of benefits. Firstly, it helps inform the QA process in validating the security of the codebase itself. Additionally, it formalizes a code security plan that can be evaluated and improved as the language and code of the code base changes and improves. It also enables new team members to maintain that security with a known level of security. |

### 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** | **Define all int variable types as either signed or unsigned** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | When operations are conducted against both signed and unsigned integer variables, conversion of signed integers to unsigned integer could result in over/underflows after operation and unexpected behaviors. |

| **Noncompliant Code** |
| --- |
| This noncompliant code block shows a signed and unsigned integer going through a subtraction operation and resulting in an overflow error |
| unsigned int a = 2;  signed int b = 3;  signed int c = a – b;  std::cout << c;  “4294967295” |

| **Compliant Code** |
| --- |
| The compliant code correctly completes the operation and does not provide an unexpected behavior |
| signed int a = 2;  signed int b = 3;  signed int c = a – b;  std::cout << c;  “-1” |

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

| **Principles(s):** Validate Input Data, Architect and Design for Security Policies, Sanitize Data Sent to Other Systems |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Low | Low | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.05 | C++ Check with native configuration for ‘int’ qualifier | High-assurance static analysis tool for C++ |
| CppCheck | 2.14 | C++ Check with no native rule for ‘int’ qualifier | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use constant variable values in place of “magic numbers”** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Having number in expressions with context can cause confusion in development and could lead to a misunderstanding as to what that number is used for. Constants variable names can clear confusion and help interpret what the code is meant to do |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses “3.14” to represent PI in finding the circumference of a circle |
| float r = 2;  float circumference = 2 \* r \* 3.14; |

| **Compliant Code** |
| --- |
| This compliant block sets the value of PI as a constant so that future code does not have to change every instance of code should more precision to the value of PI be needed and helps the developer understand what the code is doing |
| const float PI = 3.14  float r = 2  float cir = 2 \* r \* PI; |

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

| **Principles(s):** Keep It Simple, Use Effective Quality Assurance Techniques, Architect and Design for Security Policies |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Medium | Low | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14 | C++ Check with no native rule for ‘magic number’ qualifier | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |
| Clang-Tidy | 18.1.5 | C++ check utilizing ‘readability-magic-numbers’ flag | LLVM-based linter and code analyzer that checks C++ code for style, bug-prone constructs, and conformance to coding standards. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | String literals are located in read-only memory when using pointers and will result in access violation if attempts are made to modify the string literal instead of redeclaring the string literal with corrections or declares the string as an array. |

| **Noncompliant Code** |
| --- |
| The below code attempts to fix a misspelling by accessing the location in the char array for the wrong letter which results in an access violation |
| char \*str = “Hello Wotld”;  str[8] = ‘r’; |

| **Compliant Code** |
| --- |
| The complaint code below instead declares the string as an array, which then can access and write the corrected value |
| char str[] = “Hello Wotld”;  str[8] = ‘r’; |

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

| **Principles(s):** Keep It Simple, Architect and Design for Security Policies, Heed Compiler Warnings |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Medium | P2 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14 | C++ Check with native rule that detects string literals passed to non-const char\* | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |
| SonarQube (SonarC++) | 10.6 + v3.11 plugin | Web-based check requires cutom API plugin to flag assignments as potentially modifiable | CI-integrated static analysis platform with rule enforcement, code smells, and bug detection. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Validate input does not contain a potential SQL injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection can be affected when entering strings meant for operation against a SQL database, therefore, is important that any entered data intended for use against a SQL database is validated prior to use |

| **Noncompliant Code** |
| --- |
| The noncompliant code does not check for a SQL injection and simply utilizes the entered string resulting in SQL injection retrieving password data of the database |
| char input[];  cin >> intput; //”UNION SELECT username, password FROM users- -"  SQLdatabase.query(input) |

| **Compliant Code** |
| --- |
| In the complaint code, a validation function is called to look for potential SQL injections and then operate against the database. |
| char input[];  cin >> intput; //”UNION SELECT username, password FROM users- -"  if !SqlInjection(input);  SQLdatabase.query(input) |

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

| **Principles(s):** ValidateInput Data, Architect and Design for Security Policies, Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube (SonarC++) | 10.6 + v3.11 plugin | Limited support for SQ injection detection | CI-integrated static analysis platform with rule enforcement, code smells, and bug detection. |
| CodeSonar | 8.1 | Detects unsafe concatenation and unsensitized user input prior to execution code | High-end static analysis tool with deep taint tracking. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **If there is explicit object creation the must be explicit object destruction** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | When creating a new object, there is memory allocated to store that object for use until it is destroyed. If that object is never destroyed and the code base creates a new object instead of using the already existing object, a memory leak could develop. |

| **Noncompliant Code** |
| --- |
| This noncompliant code shows an object being created and which is never destroyed after use but then creates a new object resulting in a memory leak. |
| char letters[]= “a”,”b”,”c”;  struct S{  S();  void f();  };  for (int i = 0; i < sizeof(letters) -1 ; i++){  S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));  s->f();  std::free(s);  } |

| **Compliant Code** |
| --- |
| In the compliant code, the object is destroyed after use and does not stay in memory. |
| char letters[]= “a”,”b”,”c”;  struct S{  S();  void f();  };  for (int i = 0; i < sizeof(letters) -1 ; i++){  S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));  s->f();  s->~S();  std::free(s);  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques, Adhere to the Principle of Least Privilege, Architect and Design for Security Policies |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | High | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14 | C++ Check with native rule that tracks mismatched new/delete and missing delete calls | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |
| CodeSonar | 8.1 | Tracks resource flow from allocation to disposal | High-end static analysis tool with deep taint tracking. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Development assertions should be commented after development** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a useful tool to verify true conditions when developing code but could lead to unexpected behavior or reveal the structure of code that was ran. Assertions and associated status messages much be commented to ensure information about the system is not revealed. |

| **Noncompliant Code** |
| --- |
| The following code block has not commented out an assertion used in development that verified that a value from another function was as expected revealing that this part of the code has to interact with another specific part of the code base |
| String name = “Bob”;  int accountNumber = bank.getAccountNumber(name);  Assert(accountNumber == 123456);  Std::cout << name << “account number matched” << “123456”;  Bank.addMoney(100,accountNumber); |

| **Compliant Code** |
| --- |
| The compliant code comments out the assertion so that future development can be completed without rework on building new assertions |
| String name = “Bob”;  int accountNumber = bank.getAccountNumber(name);  //Assert(accountNumber == 123456);  //Std::cout << name << “account number matched” << “123456”;  Bank.addMoney(100,accountNumber); |

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

| **Principles(s):** Keep It Simple, Sanitize Data Sent to Other Systems, Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | P4 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14 | Flags excessive or redundant assert() calls but can be customized to flag assert() | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |
| CodeSonar | 8.1 | Can treat assert() as marker requiring review | High-end static analysis tool with deep taint tracking. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exception messages should not reveal details to why it was created** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception messages that are too detailed could provide information that could be utilized to circumvent security or reveal the internal operation of the code. |

| **Noncompliant Code** |
| --- |
| The exception block message in the follow try/catch block reveals that the system expects that the result will be be able to fit into an integer value |
| Int a = 3;  Long b =123412341233;  try {  long long c = a \*b;  if (c< std::numeric\_limits<int>::min() ||  c > std::numeric\_limits<int>::max()) {  throw std::out\_of\_range("The multiplication of a and b resulted in a number bigger than an integer can hold");  }  }  catch (const std::out\_of\_range& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  }  } |

| **Compliant Code** |
| --- |
| The compliant code still throws the exception but does not reveal an information on how that exception was thrown |
| Int a = 3;  Long b =123412341233;  try {  long long c = a \*b;  if (c< std::numeric\_limits<int>::min() ||  c > std::numeric\_limits<int>::max()) {  throw std::out\_of\_range("The operation resulted in a type/value mismatch");  }  }  catch (const std::out\_of\_range& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  }  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems, Practice Defense in Depth, Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Medium | Medium | P3 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1 | Tracks exceptions with potentially sensitive content | High-end static analysis tool with deep taint tracking. |
| SonarQube (SonarC++) | 10.6 + v3.11 plugin | Custom rule checks can flag detailed exception messages | CI-integrated static analysis platform with rule enforcement, code smells, and bug detection. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Close files when no longer needed** |
| --- | --- | --- |
| Input/Output | [STD-008-CPP] | Files that are no longer needed should be closed so that file locks that were created as the file was open do not prevent other processes or users from being able to access the file in question |

| **Noncompliant Code** |
| --- |
| The noncompliant code never closes the file it was operation on leading to potential issues if the file is needed later for operations. |
| void f(const std::string &fileName) {  std::fstream file(fileName);  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| This code block complies with the standard and closes the file once its operations are complete so that the file is available and no locks or errors occur when dealing with that same file again. |
| void f(const std::string &fileName) {  std::fstream file(fileName);  // ...  file.close();  std::terminate();  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques, Default Deny, Architect and Design for Security Policies, Keep It Simple |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Medium | P3 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14 | Detects when fopen() or similar are not followed with fclose() | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |
| CodeSonar | 8.1 | Detects leaked resources i.e. file handles, streams, and descriptors, reports missing fclose() calls | High-end static analysis tool with deep taint tracking. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not modify the same variable more than once in the same expression** |
| --- | --- | --- |
| Expressions | [STD-009-CPP] | Modifying a variable more than once in a single expression could result in unexpected behavior or confuse what the expression is attempting to accomplish. |

| **Noncompliant Code** |
| --- |
| This code block shows an example of i being modified twice in a way that does not have a defined order of operations |
| i = ++i + 1; |

| **Compliant Code** |
| --- |
| This block shows a compliant block that completes the two operations in separate expressions, with defined behavior for both. |
| i = i++;  i = i + 1; |

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

| **Principles(s):** Keep It Simple, Architect and Design for Security Policies, Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Low | High | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14 | Detects undefined behavior and redundant/dangerous operations | Lightweight static analysis tool for C++ code that focuses on detecting undefined behavior and code standard violations. |
| CodeSonar | 8.1 | Flags expressions that modify a variable multiple times between sequence points with C++ sequencing rules | High-end static analysis tool with deep taint tracking. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Never hard code authentication tokens/password** |
| --- | --- | --- |
| Authentication | [STD-010-CPP] | Hard coded password/tokens could be used to gained access to a resource after the credentials have been lifted from the code itself. Additionally, this prevents any ability to update a password should it become compromised and would require a recompilation of the code based to update it. |

| **Noncompliant Code** |
| --- |
| This example hard codes the database password into the connection function to access the database |
| Void dbConnect(){  char username[] = “admin”;  char password[] = “dbpassword”;  conn = driver->connect("tcp://127.0.0.1:3306", "username", "password");  }  Int main(){  dbConnect();  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant block below requires the user to input the password as a temporary variable that is destroyed out of memory once the function resolves. |
| Void dbConnect(){  char username[] = “admin”;  cin >> password;  conn = driver->connect("tcp://127.0.0.1:3306", "username", "password");  }  Int main(){  dbConnect();  return 0;  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems, Adhere to the Principle of Least Privilege, Architect and Design for Security Policies |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | P5 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1 | Detects hardcoded sensitive information in C++ source code and maps to CWE-798 and 259 | High-end static analysis tool with deep taint tracking. |
| SonarQube (SonarC++) | 10.6 + v3.11 plugin | Flags hardcoded credentials based on variable names and context and can have custom rules to look for specific strings that might have passwords | CI-integrated static analysis platform with rule enforcement, code smells, and bug detection. |

### 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.

Integrating DevSecOps into the established DevOps process employed by Green Pace will require some slight adjustments and tweaks to their present model while still maintaining the same approximate flow that Green Pace is use to.

In pre-production, the addition of security as a consideration in the planning, architecture, and design will feed into the building of the system with those security features included. On the back end of building out the system the incorporation of security testing and validations to functional and operational testing prior to deployment will ensure that the security features added are effective and in place prior to deployment.

While in production, the addition of a cybersecurity department will help verify system configurations are in line with the security policy while testing and monitoring the system looking for potential issues either from new threats or previously reported vulnerabilities. Additionally, this cybersecurity department will help respond to incidents and support the rest of the system maintainers in defining and maintaining a consistent baseline for the system with the inclusion of the security principles defined.

### 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 | Low | Low | P1 | L1 |
| STD-002-CPP | Low | Medium | Low | P1 | L1 |
| STD-003-CPP | Medium | Medium | Medium | P2 | L2 |
| STD-004-CPP | High | High | Medium | P4 | L3 |
| STD-005-CPP | Medium | Low | High | P3 | L3 |
| STD-006-CPP | Low | Low | Low | P4 | L1 |
| STD-007-CPP | Low | Medium | Medium | P3 | L2 |
| STD-008-CPP | Medium | Low | Medium | P3 | L2 |
| STD-009-CPP | High | Low | High | P4 | L3 |
| STD-010-CPP | High | Medium | High | P5 | L3 |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 | Any data that is not in active use (storage), is defined as at rest. User system passwords are a common example. While not in use, data must be encrypted to prevent plain text data exfiltration from data stores on the system. This can be storing user system passwords as a hash instead of plain text, e.g. RHEL saves passwords as a hash in the shadow file vice plain text in passwd. Drive encryption for mass storage devices such as NFS and NetApp style deployments also enforce this protection. This is policy to provide an additional layer of security for user and system data that prevents unfettered access in the case of a security breach into the system. |
| Encryption in flight | Any data that is moving either from storage to memory or to another system is considered “in flight”. Prior to use when transiting, data mush be encrypted to prevent its capture in plain text. SSH protocols are a common example of encrypting data as it transits from one side of the connection to another to prevent plain text capture. This is policy because data in flight is often the most susceptible to capture or replication. Care must be taken to ensure that data in movement is protected either from modification of capture to unauthorized entities. |
| Encryption in use | Any data being used to perform an operation is considered ‘in use’. Sensitive data such as passwords or user names, if used in operations as plain text could expose them to capture. During many authentication operations, if a plain text password is decrypted from the system store to compare against and unencrypted entered password there is risk of compromise. Instead, immediately hashing the entered password and comparing it to the hash of the password store (see ‘at rest’), prevents any plain text sensitive data and ensures that compromise cannot easily occur. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of proving that an individual is who they attest they are. This ensures that another person or entity cannot gain other authorizations on a system that they might otherwise not have. This sis accomplished via user names, passwords generally, while it is best practice to utilize multi-factor authentication where an individual must provide at least two of the following three: something they know(password), something they have(token), and something they are(biometrics). This policy applies to all users who access Green Pace systems in order to protect users from potential harm from other users and other users from that user. |
| Authorization | Authorization is the process of providing the permissions assigned to the authenticated user. These permissions are assigned IAW the principle of least privilege, i.e. a normal user will not have permission to create, edit, move, or remove files other than ones owned by that user versus a database administrator who might have permissions to their own files but also permissions to modify the database in order to fulfil their duties but no more. The level of authorization is normally determined when a new user is added to the system and verified to ensure that permission creep does not occur as the user potentially transitions to new role or responsibilities. Changes in the permission assigned of a user must be verified by a security representative and their immediate supervisor to ensure that the least amount of permission are assigned to a user. |
| Accounting | Accounting is the process of logging and providing a positive logging of every action and process, inputs, and outputs associated, e.g file access by users. This is accomplished by system, application, and security logs populated either by the system or application itself, and often additionally, other monitoring suites such as Splunk, Endgame, etc. This provides a means to ensure compliance with security policies as well as enable active monitoring and response activities for the system. This is policy because without reporting, enforcement of policy cannot occur. It is incumbent on management and security to ensure accounting is occurring to support policy enforcement and development. |

**\***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
  + 1. Data Type Standard
       1. 1. Validate Input Data – Validation of Input data is vitally important to ensure that the data being presented to the application is the correct type and does not result in undesired behavior.
       2. 3. Architect and Design for Security Policies – Ensuring that the application design includes what type should be presented prevents undesired behavior from potentially malicious inputs.
       3. 7. Sanitize Data Sent to Other Systems – Ensuring that type data is set help ensure that data sanitation can properly remove any additional data that is not necessary when transmitting to another system.
    2. Data Value Standard
       1. 4. Keep It Simple – Keeping variable data value simple enables good readability which in turn enables effective manual code reviews and prevents potential undesired interactions.
       2. 9. Use Effective Quality Assurance Techniques – In line with keeping it simple, this standard enables effective QA and provides a means to enforce standard.
       3. 3. Architect and Design for Security Policies – designing the data values to follow this standard ensures that malicious data inserted into the program cannot be operated on in a way that is malicious.
    3. String Correctness Standard
       1. 4. Keep It Simple – Complex string operations can lead to undesired behavior and maintaining a standard prevents potential harmful interactions or operations.
       2. 3. Architect and Design for Security Policies – Designing a string operation that ensures only what is intended occurs prevents undesired behavior or malicious actions.
       3. 2. Heed Compiler Warnings – Complier warnings are not recommendations and help prevent a large portion of undefined or undesired operations that might be available due to the code’s structure and operation.
    4. SQL Injection Standard
       1. 1. ValidateInput Data – Validating SQL inputs prevent undesired behavior and prevent malicious attackers from gaining potentially sensitive data from the underlying database.
       2. 3. Architect and Design for Security Policies – Designing SQL interactions to account for malicious attacks can mitigate time and effort required if refactoring the code for security is required.
    5. Memory Protection Standard
       1. 9. Use Effective Quality Assurance Techniques – QA techniques help ensure that the system resources are not overly consumed or wasted ensuring effective and efficient use of resources by the program.
       2. 6. Adhere to the Principle of Least Privilege – Ensuring that a memory allocation is left hanging can prevent it’s use by users that would not normally have access to the resources and prevents undesired behavior.
       3. 3. Architect and Design for Security Policies – Designing proper memory allocation and protections help prevent a system failure or undesired behavior.
    6. Assertions Standard
       1. 4. Keep It Simple – Assertions can support troubleshooting or debugging but rarely enhance production code and are not necessary and complicate code interactions.
       2. 7. Sanitize Data Sent to Other Systems – Assertions left in the code could potentially provide more information that is necessary for another system, exposing underlying debugging or issue resolution that could be useful to attackers.
       3. 9. Use Effective Quality Assurance Techniques – QA processes should result in a final production code that does not rely on debugging in production and must be effective in removing unnecessary code such as assertions.
    7. Exceptions Standard
       1. 7. Sanitize Data Sent to Other Systems – Verbose exceptions can provide the underlying logic present in the codebase that could be exploited by attackers or interfere with the other systems proper function.
       2. 8. Practice Defense in Depth – When combined with a proper message and other techniques exceptions can enhance security and prevent undesired behavior.
       3. 9. Use Effective Quality Assurance Techniques – Effective QA techniques will ensure that something like a verbose exception message is not provided to users and potential attackers.
    8. Input/Output Standard
       1. 9. Use Effective Quality Assurance Techniques – Effective QA techniques help enforce correct and safe IO interactions coming in and going out from the system.
       2. 5. Default Deny – Ensuring that any type of input is by default denied, protects the system from potentially malicious inputs and prevents inputs from causing undefined behavior.
       3. 3. Architect and Design for Security Policies – Designing the input/output with security mind ensures that the systems interactions outside of itself are protected and minimized to only those required.
       4. 4. Keep It Simple – Overly complex IO operations make debugging more difficult and also complicate the follow-on code that relies on those operations.
    9. Expressions Standard
       1. 4. Keep It Simple – Complex expressions are difficult to read and understand and can greatly hamper debugging and design, complicating security concerns.
       2. 3. Architect and Design for Security Policies – Designing expressions properly while considering security can reinforce other security techniques and prevent undesired behavior.
       3. 9. Use Effective Quality Assurance Techniques – QA techniques can reinforce simple and effective design considerations, enhancing readability and security.
    10. Authentication Standard
        1. 7. Sanitize Data Sent to Other Systems – Ensuring that plaintext passwords and authentication tokens is vital to protect users and services from being spoofed or other wise affected when interacting with outside systems.
        2. 6. Adhere to the Principle of Least Privilege – Ensuring that only those users requiring permissions for various authentication to other parts of the system protects the system from malicious attacks.
        3. 3. Architect and Design for Security Policies – Designing a strong authentication paradigm is vital to ensure that the system can trust that any user or service is authenticated and then authorized for the correct operations.
    11. All Standards
        1. 10. Adopt a Secure Coding Standard – Adopting a Secure coding standard is a vital step in ensure that proper QA and security controls can identify and remediate security concerns both in the initial release of software but also during ongoing DevSecOps.

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/25/2025 | Add Coding standards | Ben Cleary | [Insert text.] |
| 1.2 | 06/15/2025 | Rolled out tools/automation/mapping | Ben Cleary | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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