**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** | Definitions |
| --- | --- |
| 1. ValidateInput Data | All user input should be validated as safe before using in the execution of downstream commands. For example, user input should be sanitized with buffer overflow and SQL injection attacks in mind. |
| 1. Heed Compiler Warnings | All compiler warnings should be addressed in code before it can be considered “complete.” Compiler warnings often hint at current or future problems with the code being created, including possible security vulnerabilities. |
| 1. Architect and Design for Security Policies | A core business requirement of all code-based work product are security policies. This means that in every phase of application and/or system design, the software code being written should adhere to, at minimum, firm security policies. Where generally-accepted best practices dictate a higher degree of consideration, that posture is to be used instead. |
| 1. Keep It Simple | Keeping the architecture and layout of code-based work product as simple as possible will make the job of creating and maintaining secure software easier. Complexity is the enemy of security, as it makes testing and verifying adherence to policies and procedures more difficult than necessary. |
| 1. Default Deny | If you must ask if there is permission for a user or group of users to access information, then the default answer is “no.” Only explicit permission to access a software module or supporting dataset is valid. All software should be vigorously tested to ensure that “no user without express permission to access the subject information is able.” |
| 1. Adhere to the Principle of Least Privilege | Users or groups of users should only be able to access information that is necessary for them to complete the tasks at hand. This is, in essence, a “need-to-know” base case for every user. For example, just because a customer service representative needs to know the last four of a client’s SSN to verify their identity, does not mean they should be shown the full tax ID number anywhere within the CSR system. A user without a verified “need-to-know” this information is not someone who should have it, even if it seems benign in isolation. |
| 1. Sanitize Data Sent to Other Systems | In practice, this means ensuring that the application you are creating does not represent a security vulnerability to other systems that are either upstream or downstream in process. For example, input that you capture from a user that is then handed off to a downstream system should be checked and sanitized for command injection attempts BEFORE it is transmitted. |
| 1. Practice Defense in Depth | In the past, this was referred to as “severability” within the security framework. In practice, this is ensuring that if one layer of security is breached, other layers remain, at the very least for redundancy, so that ultimately there is no one, simple way to circumvent firm security policies. |
| 1. Use Effective Quality Assurance Techniques | Even in cases where a dedicated quality assurance team is assigned to a project, the responsibility of creating and testing for secure code belongs to all employees. Security policies dictate that static, dynamic and stress tests are completed to ensure proper security posture. In the event that generally-accepted best practices indicate additional testing, this is to be done by default as an addition to stated firm policies and procedures. |
| 1. Adopt a Secure Coding Standard | While the firm recognizes and borrows much in the way of security posture from the SEI CERT framework, it is important to achieve the highest level of security that is attainable for the development team. For example, if the firm employs subject-matter-experts (SMEs) that are capable of producing more secure code than that which is indicated by industry standards and SEI CERT, that knowledge should be employed to create a more robust system (without compromise to any firm or industry standard policies and procedures) by default. |

### 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** | **Integer Overflow Prevention** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Integer overflows occur when an arithmetic operation attempts to create a numeric value that is too large for the data type meant to store it. This can lead to unexpected behavior, possibly causing program crashes or allowing malicious code execution. |

| **Noncompliant Code** |
| --- |
| The code blindly adds two integers without checking for overflow. This can lead to wrapping effects and undefined behavior in C++. |
| int add(int a, int b) {  return a + b;  } |

| **Compliant Code** |
| --- |
| This code checks for integer overflow before performing the addition. If an overflow would occur, an exception is thrown, preventing undefined behavior. |
| #include <stdexcept>  int add(int a, int b) {  if(a > INT\_MAX - b) {  throw std::overflow\_error("Addition would cause overflow");  }  return a + b;  } |

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

| **Principles(s):** Validate Input Data: This standard maps to the principle of validating input data, ensuring that any operation using the data doesn't result in unintended consequences like overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | alpha.security.IntOverflow | Checks for potential integer overflows. |
| GCC | 11.1 | -Woverflow | Warn if overflow occurs in constant expression. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Prevention of Uninitialized Variables** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Using uninitialized variables in code can lead to unpredictable behavior since the values held by these variables are indeterminate. This is particularly relevant when dealing with data values. Uninitialized variables can also pose security risks as they might leak sensitive information from memory or be exploited by attackers to execute arbitrary code. |

| **Noncompliant Code** |
| --- |
| The code declares a local variable a but does not initialize it. As a result, the function returns an indeterminate value. |
| int function() {  int a;  return a;  } |

| **Compliant Code** |
| --- |
| The code initializes the local variable a to a value of 0. This ensures the function returns a predictable value and does not expose any sensitive information or undefined behavior. |
| int function() {  int a = 0;  return a;  } |

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

| **Principles(s):** Keep It Simple: By initializing variables when they are declared, we simplify the code by ensuring it behaves predictably, reducing chances of inadvertent errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | alpha.core.UninitializedVariable | Checks for variables that may be uninitialized when used. |
| GCC | 11.1 | -Wuninitialized | Warn if an uninitialized variable is used in the code. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Safe String Manipulation** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Correct string handling is essential in preventing various security vulnerabilities, especially those that arise from buffer overflows. By ensuring strings are correctly terminated and not manipulated in an unsafe way, we can prevent data leaks and potential code execution vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The code uses strcpy to copy a string into a buffer without checking the length of the source string. This can lead to buffer overflows if the source string is longer than the destination buffer. |
| char destination[10];  strcpy(destination, source); |

| **Compliant Code** |
| --- |
| The code uses strncpy to copy a string into a buffer with a specified maximum length. This ensures that the string doesn't exceed the buffer's capacity. |
| char destination[10];  strncpy(destination, source, sizeof(destination) - 1);  destination[sizeof(destination) - 1] = '\0'; // Ensure null termination |

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

| **Principles(s):** Validate Input Data: By checking the length of strings and ensuring they fit within the buffers they are written to, we validate the data and prevent potential overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | alpha.security.taint.TaintPropagation | Checks for potential string manipulation vulnerabilities. |
| GCC | 11.1 | -Wformat-security | Warn about format string vulnerabilities. |
| Fortify | 20.1.0 | Buffer Overrun | Detects potential buffer overflows in string operations. |
| Coverity | 9.0 | TAINTED\_STRING | Identifies potential vulnerabilities in string handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevention of SQL Injection Attacks** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL Injection is a type of attack that allows attackers to execute malicious SQL statements. Such code injections can compromise the database, manipulate or delete data, and sometimes lead to a full takeover of the host machine. Proper parameterization of SQL queries prevents this vulnerability, ensuring the integrity and confidentiality of the database. |

| **Noncompliant Code** |
| --- |
| The code constructs an SQL query by directly concatenating user input, making it susceptible to SQL injection attacks. |
| char query[200];  sprintf(query, "SELECT \* FROM users WHERE username='%s' AND password='%s'", username, password);  executeQuery(query); |

| **Compliant Code** |
| --- |
| The code uses parameterized queries, which separates SQL logic from the data, ensuring that user input is always treated as data and never as executable SQL code. |
| PreparedStatement statement = connection.prepareStatement("SELECT \* FROM users WHERE username=? AND password=?");  statement.setString(1, username);  statement.setString(2, password);  ResultSet results = statement.executeQuery(); |

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

| **Principles(s):** Validate Input Data: By parameterizing SQL queries, we ensure that all user input is treated strictly as data and not as executable code. This prevents malicious actors from injecting arbitrary SQL code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.10.0 | SQL Injection | Detects potential SQL injection vulnerabilities. |
| Fortify | 20.1.0 | SQL Injection | Identifies insecure patterns of SQL query construction. |
| Checkmarx | 9.0 | SQL\_Injection | Checks for code patterns that are vulnerable to SQL injection. |
| sqlmap | 1.5 | N/A | An open-source penetration testing tool that automates the detection and exploitation of SQL injection flaws. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Secure Memory Management and Avoidance of Buffer Overflows** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Buffer overflows are among the oldest and most dangerous vulnerabilities in software. They can lead to arbitrary code execution, crashes, or data corruption. This standard ensures that buffer boundaries are respected, and any data read into or written from buffers is managed securely. |

| **Noncompliant Code** |
| --- |
| The code reads user input without checking the length, leading to a potential buffer overflow if the input exceeds the buffer size. |
| char buffer[10];  gets(buffer); |

| **Compliant Code** |
| --- |
| The code uses fgets, which ensures that the buffer's boundary is not exceeded by user input. |
| char buffer[10];  fgets(buffer, sizeof(buffer), stdin); |

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

| **Principles(s):** Validate Input Data: Memory protection inherently involves validating the length of input data. By using functions that respect buffer boundaries, we ensure that input data does not overflow its allocated space. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Detects memory management issues and potential buffer overflows. |
| AddressSanitizer | N/A | Memory Error Detector | Instrumentation tool that detects out-of-bounds and use-after-free errors. |
| Fortify | 20.1.0 | Buffer Overflows | Identifies potential buffer overflow vulnerabilities in the code. |
| Dr. Memory | 2.3.0 | Buffer Overrun Detection | Runtime analysis tool that detects common memory errors in C/C++ programs. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Proper Use of Assertions for Code Integrity** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are utilized to ensure a program's invariants—conditions that should always be true. While invaluable for debugging and catching unexpected conditions during development, assertions should not be used as a primary security mechanism. Their misuse can lead to vulnerabilities, especially if they are disabled in production. |

| **Noncompliant Code** |
| --- |
| The code relies on an assertion to validate security-sensitive operations, a behavior that could be bypassed if assertions are disabled. |
| int withdraw(int amount) {  assert(amount >= 0);  // ... withdrawal logic ...  } |

| **Compliant Code** |
| --- |
| The code checks the condition without relying on an assertion, ensuring that the validation is not skipped even if assertions are turned off. |
| int withdraw(int amount) {  if (amount < 0) {  // handle error or return an error code  return -1;  }  // ... withdrawal logic ...  } |

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

| **Principles(s):** Defense in Depth: Assertions are an additional layer of security but should not be the only layer. Actual security checks should always be implemented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 9.0 | ASSERT\_SIDE\_EFFECT | Identifies side effects in assertions that might not always execute. |
| Clang Static Analyzer | 12.0 | Misuse of assertions | Detects misuse or over-reliance on assertions. |
| SonarQube | 8.9 | Assertion usage checks | Provides insights on proper assertion use and potential pitfalls. |
| PVS-Studio | 7.14 | V790 | Diagnoses the use of empty assertions or assertions with potential issues. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Structured and Safe Exception Handling in C++** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | In C++, exceptions provide a mechanism to handle runtime errors gracefully. A structured exception handling mechanism ensures that the program can continue its execution or terminate safely, safeguarding resource integrity and security. Misuse or lack of exception handling can lead to unpredictable behavior, resource leaks, or security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Using catch-all exceptions without specific handling or logging can mask potential issues, leading to unpredictable behavior. |
| try {  // ... some code ...  } catch (...) {  // do nothing  } |

| **Compliant Code** |
| --- |
| Exceptions should be caught more specifically, with logging to ensure that potential issues are captured for future debugging. |
| try {  // ... some code ...  } catch (const std::ios\_base::failure& e) {  std::cerr << "IO exception: " << e.what() << std::endl;  } catch (const std::exception& e) {  std::cerr << "Standard exception: " << e.what() << std::endl;  } |

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

| **Principles(s):** Robustness Principle: "Be conservative in what you do, be liberal in what you accept from others." This means that exceptions should be handled appropriately to prevent the application from crashing or acting unpredictably. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 9.0 | BAD\_CATCH | Identifies generic exception catches that can obscure potential issues in C++ code. |
| SonarQube | 8.9 | EmptyCatchBlock | Detects empty catch blocks in C++ that might suppress exceptions without handling them. |
| Clang Static Analyzer | 12.0 | Exception Safety Issues | Offers diagnostics related to potential issues in exception safety in C++. |
| PVS-Studio | 7.14 | V563, V610 | Identifies potential problems in exception handling or cases where exceptions may be thrown unexpectedly in C++. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Effective Resource Management in C++** |
| --- | --- | --- |
| Resources | STD-008-CPP | C++ does not have a garbage collector like some other languages, so the responsibility of managing memory and resources falls on the programmer. Effective resource management is essential to prevent memory leaks, avoid resource contention, and ensure optimal program performance. |

| **Noncompliant Code** |
| --- |
| Dynamically allocated memory without being freed, leading to potential memory leaks. |
| int\* arr = new int[10];  // ... some code ...  // No delete[] arr; to free memory |

| **Compliant Code** |
| --- |
| Dynamically allocated memory is properly managed using RAII principles. In this example, memory is explicitly released using delete[]. |
| int\* arr = new int[10];  // ... some code ...  delete[] arr; |

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

| **Principles(s):** RAII (Resource Acquisition Is Initialization): In C++, this principle suggests that resources should be tied to the lifespan of objects. When the object is created (initialized), it acquires the resource, and when it's destroyed, it releases the resource. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 9.0 | RESOURCE\_LEAK | Detects potential resource leaks in C++ code. |
| SonarQube | 8.9 | ResourceLeak | Identifies potential resource leaks in C++ that might not be deallocated properly. |
| Clang Static Analyzer | 12.0 | DanglingPointer, MemoryLeak | Offers diagnostics related to potential resource management issues in C++. |
| PVS-Studio | 7.14 | V773, V774 | Identifies potential problems related to dangling pointers or resources not being released. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Safe Multithreading Practices in C++** |
| --- | --- | --- |
| Multithreading | STD-009-CPP | Multithreading and concurrent execution allow programs to run tasks in parallel, potentially improving efficiency and responsiveness. However, without proper synchronization mechanisms, concurrent modifications can lead to unpredictable results, data corruption, and hard-to-trace bugs. |

| **Noncompliant Code** |
| --- |
| Accessing shared data without adequate synchronization can result in race conditions. |
| int shared\_data = 0;  void increment() {  shared\_data++;  } |

| **Compliant Code** |
| --- |
| Shared data is accessed with proper synchronization using std::mutex. |
| #include <mutex>  int shared\_data = 0;  std::mutex data\_mutex;  void increment() {  std::lock\_guard<std::mutex> lock(data\_mutex);  shared\_data++;  } |

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

| **Principles(s):** Avoiding Race Conditions: Race conditions occur when two threads access shared data concurrently. The final result depends on the order of the threads' operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 9.0 | CONCURRENCY | Detects potential concurrency-related issues in C++ code. |
| SonarQube | 8.9 | S2444, S2450, S3012 | Identifies potential race conditions or unsafe threading practices in C++. |
| Clang Static Analyzer | 12.0 | apiModeling.StdCLibraryFunctions | Provides diagnostics related to unsafe concurrency practices. |
| PVS-Studio | 7.14 | V1032, V1033 | Identifies potential issues related to unsafe multithreading practices in C++. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Utilizing Smart Pointers for Dynamic Memory Management in C++** |
| --- | --- | --- |
| Smart Pointers | STD-010-CPP | Memory leaks and improper memory management have been long-standing issues in C++. Smart pointers are constructs in the C++ Standard Library that provide automatic lifetime management, preventing many common memory-related problems and eliminating manual memory management. |

| **Noncompliant Code** |
| --- |
| Using raw pointers for dynamic memory allocation and deallocation can lead to memory leaks, double deletions, or undefined behavior. |
| int\* raw\_pointer = new int(5);  // ... some code ...  delete raw\_pointer; |

| **Compliant Code** |
| --- |
| Use std::shared\_ptr or std::unique\_ptr for dynamic memory management, eliminating the need for explicit memory deallocation. |
| #include <memory>  std::shared\_ptr<int> smart\_pointer = std::make\_shared<int>(5);  // No need for explicit delete; memory will be released automatically when no references exist. |

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

| **Principles(s):** Safety Over Raw Pointers: Using smart pointers ensures that memory is correctly managed, preventing issues like double deletions or dangling pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 9.0 | RESOURCE\_LEAK, USE\_AFTER\_FREE | Detects potential issues related to memory leaks and use-after-free scenarios. |
| SonarQube | 8.9 | S5411, S5413 | Identifies potential issues related to raw pointer usage and recommends smart pointers. |
| Clang Static Analyzer | 12.0 | cpp.NewDeleteLeaks | Diagnoses mismatches between allocations and deallocations, suggesting smart pointers. |
| PVS-Studio | 7.14 | V611, V773 | Finds potential memory leaks and suggests improvements for pointer-based management. |

### 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.

Even though there is already a DevOps process, we will integrate DevSecOps through the addition of risk assessments during planning, security code reviews during development, both static and dynamic security testing throughout all stages of development, monitoring and feedback through a combination of continuous testing of code even after deployment as well as automated reporting infrastructure that can (ideally) detect and alert the team of vulnerabilities in real time.

### 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 | High | Very High | 1 |
| STD-003-CPP | Medium | Medium | Low | Medium | 3 |
| STD-004-CPP | High | Very Likely | High | Very High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 2 |
| STD-006-CPP | Medium | Likely | Low | Medium | 3 |
| STD-007-CPP | Medium | Unlikely | Low | Low | 4 |
| STD-008-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-009-CPP | Low | Unlikely | Low | Low | 4 |
| STD-010-CPP | High | Medium | Medium | High | 2 |

### 

### 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 in rest | Encryption at rest is the process of encrypting data when it is stored, typically on physical or cloud-based storage devices. |
| Encryption at flight | Encryption in flight, or in transit, ensures data is encrypted as it travels over networks or between systems, applications, and users. |
| Encryption in use | Encryption in use pertains to the protection of data being actively processed or used in applications, caches, or system memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of a user, system, or application trying to access a resource. It ensures that entities are who they claim to be. |
| Authorization | Once authenticated, authorization determines what actions the user, system, or application is allowed to perform. It defines the level of access and permissions. |
| Accounting | Accounting keeps track of user activities on a system, ensuring there's a record of who did what and when. |

**\***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 | 08/12/2023 | Revised Policy | Max Freeman | Board of Directors |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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