**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 all user inputs are validated to prevent invalid or malicious data. For example, inputs should match expected formats and ranges. |
| 1. Heed Compiler Warnings | Address all compiler warnings to prevent potential vulnerabilities caused by ignored errors or misconfigurations. |
| 1. Architect and Design for Security Policies | Incorporate security considerations in the early stages of software design to minimize risks and ensure consistent implementation. |
| 1. Keep It Simple | Use straightforward and clear designs to reduce complexity, making systems easier to understand and maintain. |
| 1. Default Deny | Set default system behaviors to deny access unless explicitly permitted. |
| 1. Adhere to the Principle of Least Privilege | Assign only the minimum necessary access rights to users and processes to perform tasks. |
| 1. Sanitize Data Sent to Other Systems | Filter and sanitize data before transmitting it to external systems to prevent vulnerabilities like SQL injection. |
| 1. Practice Defense in Depth | Employ multiple layers of security controls to protect against various threats and attack vectors. |
| 1. Use Effective Quality Assurance Techniques | Regularly test and review code to detect and mitigate vulnerabilities early. |
| 1. Adopt a Secure Coding Standard | Follow established coding standards to ensure uniformity and reduce security flaws in development. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Obey the One-Definition Rule  The One-Definition Rule (ODR) ensures that each object, function, and type is defined only once across all translation units. While some scenarios, such as platform-specific definitions, require unique headers, inadvertent duplicate definitions can lead to linking errors or undefined behavior. |

| **Noncompliant Code** |
| --- |
| This example shows two different translation units defining the same class S with different structures, which causes a conflict. |
| // a.cpp  struct S { int a; };    // b.cpp  class S { public: int a; }; |

| **Compliant Code** |
| --- |
| Using a header file ensures that the definition of S is consistent across all files. |
| // S.h  struct S { int a; };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):**  **Validate Input Data:** Ensures consistent data types across modules.  **Architect and Design for Security Policies:** Encourages planned and systematic definition of objects, reducing risk. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium (common in larger codebases) | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.0 | Cpp:S1944 | Detects violations of the One-Definition Rule across files, ensuring consistency in definitions and preventing accidental duplication. |
| Clang Static Analyzer | 15.0 | ODRViolation | Flags potential ODR violations, especially in complex builds with multiple translation units. |
| PVS-Studio | 7.20 | V1048 | Analyzes C++ code for potential ODR violations and redundancy in object definitions. |
| GCC Compiler (Warnings Enabled) | 12.2.0 | -Wodr | Emits warnings for potential ODR violations during compilation, especially for projects with multiple translation units. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do Not Read Uninitialized Memory  Accessing uninitialized memory can lead to unpredictable behavior since the memory might contain garbage values. This can result in vulnerabilities if the program relies on uninitialized values, potentially exposing sensitive data or causing crashes. |

| **Noncompliant Code** |
| --- |
| Here, i is used without being initialized, which can lead to undefined behavior. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| This version initializes i to ensure it holds a defined value before being used. |
| #include <iostream>  void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):**    Validate Input Data: Ensures all variables and memory are properly initialized before use.    Heed Compiler Warnings: Addressing compiler warnings helps to detect uninitialized memory early.    Use Effective Quality Assurance Techniques: Identifies potential issues through static analysis and runtime checks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.3 | UNINIT | Identifies uninitialized variable usage. |
| Clang Static Analyzer | 15.0 | UninitializedValues | Flags uninitialized variables that may result in undefined behavior during runtime. |
| Polyspace Bug Finder | R2023a | CERT C++: EXP53-CPP | Detects non-initialized variables, pointers, and other memory access issues. |
| PVS-Studio | 7.20 | V6001 | Highlights cases of uninitialized variable usage in various scenarios. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Avoid Creating std::string from Null Pointers  Creating a std::string from a null pointer leads to undefined behavior. Always validate pointers before using them to initialize strings. |

| **Noncompliant Code** |
| --- |
| This code does not check the result of std::getenv() before creating a std::string. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // Use tmp  }  } |

| **Compliant Code** |
| --- |
| Validate the pointer returned by std::getenv() before constructing a std::string. |
| #include <cstdlib>  #include <string>    void f() {  const char\* tmpPtr = std::getenv("TMP");  std::string tmp(tmpPtr ? tmpPtr : "");  if (!tmp.empty()) {  // Use tmp  }  } |

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

| **Principles(s):**  Validate Input Data: Ensures that inputs (pointers) are checked for null values before usage.    Sanitize Data Sent to Other Systems: Avoids propagating invalid or undefined data, ensuring system integrity.    Heed Compiler Warnings: Enables early detection of issues during development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0 | NullPointerCheck | Checks for null pointer dereference issues. |
| Coverity | 2023.3 | NPD | Identifies null pointer dereference scenarios during analysis. |
| PVS-Studio | 7.20 | V501 | Highlights potential null pointer dereference cases in C++. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-STR51 | Ensures compliance with CERT standards for string handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection  SQL injection vulnerabilities occur when untrusted user inputs are concatenated directly into SQL queries, allowing attackers to manipulate the database. Using parameterized queries and input sanitization prevents these attacks. |

| **Noncompliant Code** |
| --- |
| This code concatenates untrusted input into an SQL query, making it vulnerable to injection. |
| std::string uName = getRequestString("username");  std::string uPass = getRequestString("userpassword");  std::string sql = "SELECT \* FROM Users WHERE Name = '" + uName + "' AND Pass = '" + uPass + "'"; |

| **Compliant Code** |
| --- |
| This code uses a parameterized query to securely handle user inputs. |
| std::string username, userpassword;  std::cin >> username >> userpassword;  std::string sql = "SELECT \* FROM Users WHERE Name = ? AND Pass = ?";  PreparedStatement pStmt = db.prepareStatement(sql);  pStmt.setString(1, username);  pStmt.setString(2, userpassword); |

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

| **Principles(s):**  Validate Input Data: Ensures that user inputs are properly sanitized to prevent malicious entries.    Sanitize Data Sent to Other Systems: Prevents untrusted data from compromising system integrity.    Adopt a Secure Coding Standard: Promotes use of practices that inherently reduce risks of injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.3 | SQLI | Detects SQL injection vulnerabilities in query construction. |
| Fortify | 23.1 | SQL\_Injection | Flags potential SQL injection flaws in applications during static analysis. |
| SonarQube | 9.9 | cpp:S3649 | Identifies unsafe database query practices, focusing on preventing injection attacks. |
| Chackmarx | 2023.2 | SQL\_Injection | Highlights SQL injection risks using advanced taint analysis. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do Not Access Freed Memory  Accessing memory after it has been freed can result in undefined behavior, crashes, or exploitable vulnerabilities. Proper memory management prevents these risks. |

| **Noncompliant Code** |
| --- |
| This code dereferences a pointer after its memory has been freed. |
| struct S {  void f();  };  void g() {  S\* s = new S;  delete s;  s->f(); // Undefined behavior  } |

| **Compliant Code** |
| --- |
| Memory is accessed only while it is allocated, and proper cleanup is performed. |
| struct S {  void f();  };  void g() {  S\* s = new S;  s->f();  delete s;  } |

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

| **Principles(s):**  Practice Defense in Depth: Uses multiple validation layers to protect memory management.    Validate Input Data: Ensures memory requests are valid and checked.    Adopt a Secure Coding Standard: Encourages systematic and safe memory allocation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind (Memcheck) | 3.20.0 | Memory Leak/Invalid Access | Detects memory leaks, invalid memory access, and use-after-free errors. |
| Coverity | 2023.3 | MEM\_LEAK | Identifies improper memory allocation and deallocation. |
| Clang Static Analyzer | 15.0 | MallocChecker | Detects memory management issues like double free and leaks. |
| PVS-Studio | 7.20 | V730 | Flags unsafe memory allocation patterns. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use Static Assertions to Test Constant Expressions  Static assertions evaluate conditions at compile-time, preventing runtime errors and ensuring that key properties hold for the program's behavior. |

| **Noncompliant Code** |
| --- |
| This code uses assert() for a compile-time condition, which is evaluated at runtime. |
| #include <assert.h>  struct Timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  int func() {  assert(sizeof(Timer) == sizeof(unsigned char) + 2 \* sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| Use static\_assert for compile-time validation. |
| struct Timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  static\_assert(sizeof(Timer) == sizeof(unsigned char) + 2 \* sizeof(unsigned int),  "Structure must not have padding."); |

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

| **Principles(s):**  Validate Input Data: Ensures program assumptions hold true.    Practice Defense in Depth: Adds a layer of validation to enforce code correctness.    Use Effective Quality Assurance Techniques: Helps detect issues during development phases. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.20 | V001 | Flags improper assertion use or redundancy. |
| Coverity | 2023.3 | ASSERTION | Detects cases where assertions may fail or are misused. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC20 | Ensures assertions follow secure coding guidelines. |
| Clang Static Analyzer | 15.0 | NullPointerCheck | Checks for null pointer dereference issues. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Do Not Abruptly Terminate the Program  Abruptly terminating a program using abort() or exit() can lead to resource leaks and undefined behavior if destructors or cleanup functions are not properly called. Handling exceptions ensures the program can recover or gracefully shut down. |

| **Noncompliant Code** |
| --- |
| This code uses std::terminate() indirectly when f() throws an exception during program termination. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  std::atexit(f);  return 0;  } |

| **Compliant Code** |
| --- |
| This code wraps the call to throwing\_func() in a try-catch block, ensuring proper exception handling. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() {  try {  throwing\_func();  } catch (...) {  // Handle error  }  }  int main() {  std::atexit(f);  return 0;  } |

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

| **Principles(s):**  Adopt a Secure Coding Standard: Encourages the use of specific and meaningful exception types.    Practice Defense in Depth: Ensures exceptions are handled appropriately to maintain stability.    Use Effective Quality Assurance Techniques: Validates exception handling through testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.3 | EXCEPTION | Detects improper exception handling and potential unhandled exceptions. |
| PVS-Studio | 7.20 | V1074 | Highlights issues with unhandled or redundant exception handling. |
| Clang Static Analyzer | 15.0 | ExceptionSafetyChecker | Flags unsafe or missing exception handling cases. |
| Polyspace Bug Finder | R2023a | CERT\_CPP-ERR51 | Ensures adherence to exception handling best practices. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | Write Constructor Initializers in Canonical Order  The initialization order in a constructor should match the declaration order of class members to ensure deterministic behavior and avoid using uninitialized data. |

| **Noncompliant Code** |
| --- |
| This code initializes dependsOnSomeVal before someVal, which is undefined behavior because the declaration order is not respected. |
| class C {  int dependsOnSomeVal;  int someVal;  public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| Reorder the class members so they are initialized in the correct order. |
| class C {  int someVal;  int dependsOnSomeVal;  public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):**  Adopt a Secure Coding Standard: Ensures use of cryptographically secure methods for random generation.    Practice Defense in Depth: Prevents predictable outcomes in randomization processes.    Sanitize Data Sent to Other Systems: Ensures randomness quality meets security standards. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.3 | RANDOM | Detects insecure random number generation methods. |
| SonarQube | 9.9 | cpp:S2245 | Identifies the use of weak random number generators. |
| PVS-Studio | 7.20 | V1025 | Highlights instances where rand() or other weak RNG methods are used. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MSC50-a | Ensures compliance with secure random number generation practices. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-009-CPP | Use Valid Iterator Ranges  Using invalid iterator ranges, such as comparing iterators from different containers or passing invalid ranges to algorithms, leads to undefined behavior. |

| **Noncompliant Code** |
| --- |
| This code passes iterators in the wrong order to std::for\_each, causing undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>  void f(const std::vector<int>& c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| Ensure the iterators represent a valid range. |
| #include <algorithm>  #include <iostream>  #include <vector>  void f(const std::vector<int>& c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):**  Validate Input Data: Ensures that input is checked for validity before use.    Practice Defense in Depth: Adds a layer of protection against potential exploits.    Adopt a Secure Coding Standard: Encourages the use of proper bounds-checking techniques. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.3 | OUT\_OF\_BOUNDS | Detects out-of-bounds access in arrays and data structures. |
| PVS-Studio | 7.20 | V112 | Highlights potential out-of-bounds access in indexed arrays. |
| Clang Static Analyzer | 15.0 | BoundsChecker | Flags instances of out-of-bounds memory access. |
| Polyspace Bug Finder | R2023a | CERT\_CPP-ARR36 | Ensures compliance with secure array handling practices. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-010-CPP | Do Not Access Objects Outside Their Lifetime  Accessing objects outside their defined lifetime can result in undefined behavior, memory corruption, or crashes. Properly manage object lifetimes to avoid these issues. |

| **Noncompliant Code** |
| --- |
| This code dereferences a pointer before it is properly initialized. |
| struct S {  void mem\_fn();  };  void f() {  S\* s;  s->mem\_fn(); // Undefined behavior  } |

| **Compliant Code** |
| --- |
| Allocate memory for the object before accessing it. |
| struct S {  void mem\_fn();  };  void f() {  S\* s = new S;  s->mem\_fn();  delete s;  } |

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

| **Principles(s):**  Practice Defense in Depth: Reduces exposure of sensitive data by ensuring secrets are not stored in source code.    Adopt a Secure Coding Standard: Promotes practices that secure sensitive information.    Validate Input Data: Ensures secrets retrieved from storage are valid and sanitized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GitHub Secret Scanning | Latest | Secret Detection | Detects hard-coded secrets in repositories and alerts developers. |
| SonarQube | 9.9 | cpp:S2068 | Identifies hard-coded credentials in source code. |
| Coverity | 2023.3 | HARDCODED\_SECRET | Detects sensitive data hard-coded in source files. |
| TruffleHog | 3.0 | Entropy Search | Searches for high-entropy strings in code that might represent secrets. |

### 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.

In the planning phase, automation can be used to integrate threat modeling and secure design practices. Tools like SonarQube and Checkmarx can analyze architecture diagrams and early code drafts for potential vulnerabilities, ensuring that security considerations are embedded from the start. During the creation phase, integrated development environment (IDE) plugins such as Coverity can provide real-time code analysis, helping developers adhere to secure coding standards as they write code.

In the verification phase, static analysis tools like PVS-Studio and dynamic analysis tools such as OWASP ZAP play a critical role in scanning for vulnerabilities. These tools ensure compliance with coding standards before the project moves to testing phases. As the project approaches the pre-production phase, additional testing such as chaos testing and input fuzzing can be conducted using tools like Valgrind. These tests simulate unexpected conditions and verify the system's resilience and security measures against edge cases.

Once the project reaches the release phase, secure software signing practices can be implemented to ensure the integrity of the code during deployment. Tools like GitHub's Secret Scanning can also be employed to detect and mitigate hard-coded credentials or secrets before release. During the monitoring phase, runtime security analysis tools like Axivion Bauhaus Suite can continuously monitor the application to enforce compliance. Logging and alerting systems can notify teams of anomalies or non-compliance in production environments, enabling swift responses to potential threats.

Finally, in the respond and adapt phase, Green Pace can leverage tools such as TruffleHog and Polyspace Bug Finder to identify new threats and update tools and configurations accordingly. Automated incident response workflows can be established to effectively mitigate risks and ensure continuous improvement of the security posture.

### 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 | Medium | Low | High | 4 |
| STD-003-CPP | High | Low | Low | Medium | 3 |
| STD-004-CPP | High | High | Medium | High | 1 |
| STD-005-CPP | High | Medium | Medium | High | 1 |
| STD-006-CPP | Medium | Low | Low | Medium | 3 |
| STD-007-CPP | High | Medium | Medium | High | 2 |
| STD-008-CPP | High | Medium | Low | High | 1 |
| STD-009-CPP | High | High | Medium | High | 1 |
| STD-010-CPP | Critical | High | Low | High | 1 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the protection of data stored on physical media, such as hard drives or cloud storage. This policy applies to databases, backups, and file systems, ensuring that data remains secure even if physical devices are compromised. Common methods include AES-256 encryption and secure key management solutions. Encryption at rest is essential to comply with regulations like GDPR and HIPAA, protecting sensitive information from unauthorized access. |
| Encryption in flight | Encryption in flight involves securing data as it travels across networks to prevent interception by malicious actors. This policy applies to communications between servers, APIs, and user devices. Secure protocols such as HTTPS, TLS 1.3, and VPNs should be implemented to safeguard data integrity and confidentiality during transmission. This encryption method prevents man-in-the-middle (MITM) attacks and ensures secure user interactions. |
| Encryption in use | Encryption in use addresses the protection of data actively being processed or stored in volatile memory, such as RAM. This policy is crucial for safeguarding sensitive information, like cryptographic keys, during runtime operations. Tools such as Intel SGX and AWS Nitro Enclaves can create secure environments for processing critical data. Encryption in use ensures that data remains protected even during execution. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of users accessing the system. This policy applies to all user logins, requiring strong methods such as multi-factor authentication (MFA) and secure password policies. By ensuring only authorized individuals can access the system, authentication mitigates risks of unauthorized access and data breaches. |
| Authorization | Authorization defines user permissions based on roles and responsibilities, ensuring adherence to the principle of least privilege. This policy applies to all actions, such as accessing files, making database changes, or adding new users. Role-based access control (RBAC) and OAuth are examples of secure authorization mechanisms that enforce strict access management. |
| Accounting | Accounting involves the logging and monitoring of user activities within the system. This policy applies to user actions, such as file access, database modifications, and system changes. Comprehensive logs created using tools like Splunk or ELK Stack provide traceability and accountability, aiding in compliance and forensic investigations. |

**\***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
* Standard: STD-001-CPP (Data Type Correctness)

Principles Applied:

Validate Input Data

Use Effective Quality Assurance Techniques

Justification:

Validating input ensures that data types conform to expected formats, preventing undefined behavior. By integrating quality assurance techniques, such as static analysis, developers can detect potential issues early, enhancing system robustness.

* Standard: STD-002-CPP (Data Value Correctness)

Principles Applied:

Validate Input Data

Sanitize Data Sent to Other Systems

Justification:

Input validation guarantees the correctness of values, reducing the risk of data corruption. Sanitizing outbound data ensures compliance with system protocols, preventing vulnerabilities like injection attacks.

* Standard: STD-003-CPP (String Correctness)

Principles Applied:

Validate Input Data

Use Effective Quality Assurance Techniques

Justification:

By validating string inputs and employing automated quality checks, this standard prevents null pointer exceptions and other string-related vulnerabilities.

* Standard: STD-004-CPP (Prevent SQL Injection)

Principles Applied:

Sanitize Data Sent to Other Systems

Practice Defense in Depth

Justification:

Sanitizing inputs eliminates the risk of SQL injection, while implementing defense-in-depth techniques ensures multiple layers of security.

* Standard: STD-005-CPP (Memory Protection)

Principles Applied:

Practice Defense in Depth

Adopt a Secure Coding Standard

Justification:

Defensive programming practices and adherence to secure coding guidelines protect against memory leaks, buffer overflows, and related vulnerabilities.

* Standard: STD-006-CPP (Assertions)

Principles Applied:

Use Effective Quality Assurance Techniques

Heed Compiler Warnings

Justification:

Assertions ensure that code invariants are maintained during development. Addressing compiler warnings proactively prevents runtime errors.

* Standard: STD-007-CPP (Exceptions)

Principles Applied:

Adopt a Secure Coding Standard

Plan Ahead for Error Handling

Justification:

Following secure coding practices ensures robust exception handling. Planning for errors mitigates system crashes and unexpected behavior.

* Standard: STD-008-CPP (Random Number Generators)

Principles Applied:

Adopt a Secure Coding Standard

Sanitize Data Sent to Other Systems

Justification:

Using secure random number generation avoids predictable outputs. Sanitizing random outputs ensures compliance with system protocols.

* Standard: STD-009-CPP (Input Bounds)

Principles Applied:

Validate Input Data

Practice Defense in Depth

Justification:

Validating input bounds protects against buffer overflows, and layering defense mechanisms reduces risk of exploitation.

* Standard: STD-010-CPP (Hard-Coded Secrets)

Principles Applied:

Keep Sensitive Data Private

Practice Defense in Depth

Justification:

Avoiding hard-coded secrets safeguards sensitive data. Applying defense-in-depth ensures that additional measures, like secure key management, protect secrets comprehensively.

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 |  |
| 1.2 | 12/7/2024 | Finish Project 1 | David Hughes |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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