**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 | Input validation ensures that data received by an application is properly checked before being processed. This helps prevent vulnerabilities like SQL injection, buffer overflows, and cross-site scripting by rejecting malicious input. |
| 1. Heed Compiler Warnings | Compiler warnings highlight potential security issues, such as buffer overflows and uninitialized variables. These warnings can help minimize risks |
| 1. Architect and Design for Security Policies | Security should be built into the software from the start. Design applications with clear rules for authentication, access control, and data protection to reduce risks. |
| 1. Keep It Simple | Simple systems can be easy to secure and maintain, Complex design increase potential security flaws. |
| 1. Default Deny | System should adapt a deny by default approach, granting access to only authorized personal. |
| 1. Adhere to the Principle of Least Privilege | Users, processes, and systems should only have the minimal level of access required to perform their tasks. This reduces the potential impact of a security breach by limiting the privileges available to an attacker. |
| 1. Sanitize Data Sent to Other Systems | Any data transmitted to external systems, databases, or APIs must be sanitized to remove potentially harmful content. This prevents command injection attacks, unintended execution, and data corruption. |
| 1. Practice Defense in Depth | Defense in Depth (DiD) is a cybersecurity strategy that involves implementing multiple layers of security controls to protect systems and data. The depth of security required depends on the type of data being safeguarded. |
| 1. Use Effective Quality Assurance Techniques | Before deployment Secure coding practices should be reinforced with rigorous testing, including static code analysis, and security-focused code reviews, to identify and fix vulnerabilities. Vu |
| 1. Adopt a Secure Coding Standard | Following industry-recognized secure coding standards, such as SEI CERT C++, helps ensure best practices for writing secure code. This minimizes the risk of common vulnerabilities and enforces consistency across the development team. |

### 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] | Avoid Ambiguous Declarations (Vexing Parse) |

| **Noncompliant Code** |
| --- |
| Ambiguous syntax can cause unintended interpretations, leading to unexpected behavior. |
| static std::mutex m;  static int shared\_resource;  void increment\_by\_42() {  std::unique\_lock<std::mutex>(m); // Interpreted as a function declaration, NOT locking the mutex.  shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| Explicitly define objects to avoid ambiguity. |
| static std::mutex m;  static int shared\_resource;  void increment\_by\_42() {  std::unique\_lock<std::mutex> lock(m); // Clearly declares a lock object.  shared\_resource += 42;  } |

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

| **Principles(s):** Understandability: Code must be unambiguous to ensure that its meaning is clear to both humans and compilers.  Correctness: Prevents the compiler from misinterpreting the code as a function declaration.  Fail-Safe Defaults: Ensures the code does not rely on compiler heuristics to resolve ambiguity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.STRUCT.DECL.FNEST | Detects nested and ambiguous function declarations |
| Helix QAC | 2024.4 | C++1109, C++2510 | Catches ambiguous declarations and improper syntax |
| Klocwork | 2024.4 | CERT.DCL.AMBIGUOUS\_DECL | Flags confusing or ambiguous declaration syntax |
| LDRA tool suite | 9.7.1 | 296 S | Partial coverage; highlights ambiguous constructs |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Initialize Variables Before Use |

| **Noncompliant Code** |
| --- |
| Using uninitialized variables results in undefined behavior. |
| void f() {  int i; // Uninitialized variable  std::cout << i; // Undefined behavior  } |

| **Compliant Code** |
| --- |
| Always initialize variables before use to ensure predictable behavior. |
| void f() {  int i = 0; // Properly initialized  std::cout << i; // Safe output  } |

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

| **Principles(s):** Validate Input Data – Ensures variables hold defined, safe values before use.  Keep It Simple – Simplifies debugging and avoids unexpected behavior.  Adopt a Secure Coding Standard – Aligns with secure practices to prevent vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Unintialzed-read | Partially checked |
| Clang | 3.9 | -Wuninitialized clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| CodeSonar | 9.0p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local Uninitialized variable |
|  |  |  |  |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Partially implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Prevent Buffer Overflows and Ensure Null Termination |

| **Noncompliant Code** |
| --- |
| Using unbounded input can cause buffer overflow, leading to security vulnerabilities. |
| void f() {  char buf[12];  std::cin >> buf; // Potential buffer overflow  } |

| **Compliant Code** |
| --- |
| Using std::string eliminates the need for manual buffer management, preventing overflows and truncation. |
| void f() {  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo; // Safe input handling  } |

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

| **Principles(s):** Validate Input Data – Ensures input is checked to prevent unexpected buffer overflow vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| RuleChecker | 22.10 | stream-input-char-array | Partially checked |
| CodeSonar | 9.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator Buffer overrun Type overrun |
| Helix QAC | 2024.4 | C++5216  DF2835, DF2836, DF2839, |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR50-b CERT\_CPP-STR50-c CERT\_CPP-STR50-e CERT\_CPP-STR50-f CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| Malicious input can manipulate the query, leading to unauthorized access or data manipulation. |
| String Name = getRequestString(“username”);  String Pass = getRequestString(“userPassword’);  Sql = “SELECT \* FROM Users WHERE Name = “ + Name + “ ’ AND PASS= ‘ “ + Pass + “ ’ ”; |

| **Compliant Code** |
| --- |
| Use parameterized queries or prepared statements to securely handle user input, preventing SQL injection. |
| PreparedStatement pStmt = PreparedStatement();  cin >> username;  cin >> userpassword;  sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username, userpassword; |

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

| **Principles(s):** Keep It Simple – Using parameterized queries or prepared statements simplifies and secures database interactions, reducing error-prone manual query construction. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors (see Chapter 8) |
| CodeSonar | 9.0p0 | JAVA.IO.INJ.SQL | SQL injection |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| The following code does not check whether memory allocation succeeded, which can lead to undefined behavior if (new) fails. |
| void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| Using std::nothrow ensures that the function properly detects and handles allocation failures instead of throwing an exception. |
| void f(const int \*array, std::size\_t size) noexcept {    int \*copy = new (std::nothrow) int[size];    if (!copy) {      // Handle error      return;    }    std::memcpy(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

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

| **Principles(s):** Heed compiler warnings – Helps catch potential memory issues flagged during compilation. Default deny – Assumes allocation can fail and denies further execution unless memory is safely acquired. Adhere to the principle of least privilege – Limits memory usage to only what's necessary, reducing risk. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Polyspace Bug Finder | R2024b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| Using assert() to check a constant expression in a memory-mapped structure is not appropriate because assertions are typically used for runtime checks, not compile-time checks. |
| #include <assert.h>  struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| We can use static\_assert to enforce compile-time checks in type-safe way. |
| #include <type\_traits>  struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):** Heed Compiler Warnings 10: Adopt a Secure Coding Standard |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 9.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| In this example neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called, resulting in abnormal termination. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main() function handles all exceptions. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** Default deny – Assumes functions may fail and catches all exceptions to prevent undefined behavior.  Use effective quality assurance techniques – Ensures robust error handling by validating that all thrown exceptions are managed properly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| Polyspace Bug Finder | R2024b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | [STD-008-CPP] | Ensure proper management of file resources |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is created, which calls std::basic\_filebuf<T>::open(). However, if std::terminate() is called before file.close(), the file is not properly closed because std::terminate() calls std::abort(), which skips destructor calls.- |
| #include <exception>  #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate(); // File resources are not closed properly  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is explicitly called before std::terminate() to ensure the file resources are properly closed |
| #include <exception>  #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close(); // Properly close the file  if (file.fail()) {  // Handle error  }  std::terminate(); // File resources are closed properly  } |

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

| **Principles(s):** Use effective quality assurance techniques – Validates proper file handling and resource cleanup during testing.  Adopt a secure coding standard – Ensures consistent use of resource management best practices like explicitly closing files. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024b | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| CodeSonar | 9.0p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2024.4 | DF4786, DF4787, DF4788 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | Value returning Functions Must Return a Value from All Exit Paths |

| **Noncompliant Code** |
| --- |
| This noncompliant example does not return a value when the input is positive, leading to undefined behavior |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a value is returned for both negative and positive inputs, ensuring that all code paths return a value |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

| **Principles(s):** Heed compiler warnings helps detect potential resource leaks at compile time. Use effective quality assurance techniques validates proper file handling and resource cleanup during testing. Adopt a secure coding standard ensures consistent use of resource management best practices like explicitly closing files. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probably | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.STRUCT.MRS  LANG.STRUCT.NVNR | Missing return statement Non-void noreturn, |
| Astrée | 22.10 | return-implicit | Fully checked |
| LDRA tool suite | 9.7.1 | 2 D, 36 S | Fully implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MSC52-a | All exit paths from a function, except main(), with non-void return type shall have an explicit return statement with an expression |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-010-CPP] | Do not delete a polymorphic object without a virtual destructor |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, b is a polymorphic pointer type whose static type is Base\* and whose dynamic type is Derived\*. When b is deleted, it results in undefined behavior because Base does not have a virtual destructor. |
| struct Base {  virtual void f();  };  struct Derived : Base {};  void f() {  Base \*b = new Derived();  // ...  delete b; // Undefined behavior  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the Base class is defined with a virtual destructor. This ensures that when delete is called on a base class pointer pointing to a derived class object, the destructor of the derived class is called, avoiding undefined behavior. |
| struct Base {  virtual ~Base() {} // Virtual destructor  virtual void f();  };  struct Derived : Base {};  void f() {  Base \*b = new Derived();  // ...  delete b; // Properly calls Derived destructor  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | non-virtual-public-destructor-in-non-final-class | Partially checked |
| CodeSonar | 9.0p0 | LANG.STRUCT.DNVD | delete with Non-Virtual Destructor |
| LDRA tool suite | 9.7.1 | 303 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-OOP52-a | Define a virtual destructor in classes used as base classes which have virtual functions |

### 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.

[Insert your written explanations here.]

### 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 | Probably | Medium | High | 1 |
| STD-003-CPP | Hight | Likely | Medium | High | 1 |
| STD-004-CPP | Hight | Likely | Medium | High | 1 |
| STD-004-CPP | Hight | Likely | Medium | High | 1 |
| STD-005-CPP | Hight | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | High | Low | 2 |
| STD-007-CPP | Low | Probably | Medium | Low | 3 |
| STD-008-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-009-CPP | Medium | Probably | Medium | Low | 2 |
| STD-010-CPP | Low | Unlikely | Low | Low | 2 |
| [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 | What it is: Protecting data stored on physical or virtual devices using encryption.  How to apply: Encrypt all stored sensitive data using strong encryption algorithms. This includes data on hard drives, servers, mobile devices, and cloud storage.  Why it should be used: Prevents unauthorized access to data in case of device theft or security breaches. |
| Encryption in flight | What it is: Securing data while it is being transmitted between systems.  How to apply: Use secure transmission protocols such as HTTPS, TLS, and VPNs to encrypt data during communication.  Why it should be used: Prevents interception or tampering of data while it is in transit across networks. |
| Encryption in use | What it is: Protecting data while it is being processed in memory.  How to apply: Implement access controls, use trusted execution environments, and ensure security during real-time data operations.  Why it should be used: Protects sensitive data from unauthorized access or exposure during use by applications or system processes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | What it is: Verifying the identity of a user or system.  How to apply: Use strong authentication methods such as multi-factor authentication, biometrics, or secure passwords.  Why it should be used: Ensures that only authorized users can access systems and data, reducing the risk of impersonation or unauthorized access. |
| Authorization | What it is: Determining what actions or resources a user is allowed to access after authentication.  How to apply: Implement role-based or attribute-based access controls to limit access to only necessary functions or data.  Why it should be used: Helps enforce the principle of least privilege and minimizes exposure to sensitive resources. |
| Accounting | What it is: Tracking and logging user activities within a system.  How to apply: Maintain audit logs that record login attempts, data access, and system changes. Monitor logs regularly for suspicious activity.  Why it should be used: Supports accountability, and ensures compliance with regulatory requirements. |

**\***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 |  |
| [Insert text.] | 4/15/2025 | Final Template | Mohamed Elamrazougui |  |
| [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 |