**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 | When inputting data acquired by untrusted sources, you should validate, and test all data accordingly. By following this principle, most software vulnerabilities will be eliminated from your work. |
| 1. Heed Compiler Warnings | Use the highest warning level available to the compiler you are using. Eliminate all warnings thrown by the compiler and use analysis tools to eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | When designing software, decide if certain privileges will be called at certain times, and create your software with this in mind to eliminate potential security flaws. |
| 1. Keep It Simple | Keep your design as simple as possible. When system design becomes more complex, there are increased chances of security flaws or design flaws. |
| 1. Default Deny | Design your software on permission rather than exclusion. Always have denial be the base of your system, where users must show permission for admission, rather than the opposite. |
| 1. Adhere to the Principle of Least Privilege | Design your software to give the least amount of privilege needed to do a task and for the minimum amount of time possible. This makes it harder for people to exploit privilege authority. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data being sent to other subsystems to prevent attacks such as SQL, command, and other injection attacks. |
| 1. Practice Defense in Depth | Use multiple layers of defense as fail-safes. If one layer of defense is eroded, there is another layer of defense ready to fight off the attack. |
| 1. Use Effective Quality Assurance Techniques | Using effective quality assurance techniques such as source code audits, penetration testing, etc. allows for identifying and eliminating vulnerabilities. Also, using independent security reviews also allows for more vulnerabilities to be identified and fixed. |
| 1. Adopt a Secure Coding Standard | Practice and develop your coding style to use secure coding. This will help you write more secure software with less vulnerabilities. |

### 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** | **Declarations and Initialization (DCL)** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | All non-inline functions and variables within a C++ program needs to contain exactly one definition. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the definitions may look equivalent, but they are not defined using the same sequence of tokens. |
| // a.cpp  struct S {  int a;  };  // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| In the compliant code, we can see that the programmer designed a header file for the .cpp files to call, keeping definitions of functions proper. |
| // 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):** Adopt a secure coding standard. By having a secure coding standard you adhere to, your code will not run into as many errors, issues or vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Expressions (EXP)** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Every object has a lifetime. Do not use the object or a pointer to an object outside of its lifetime. |

| **Noncompliant Code** |
| --- |
| The noncompliant code has a pointer to an object that calls it before the pointer’s lifetime. |
| struct S {    void mem\_fn();  };    void f() {    S \*s;    s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| In the compliant code, storage is obtained prior to calling the pointer. |
| 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):** Heed compiler warnings. In the noncompliant code above, there should be compiler warnings, giving errors or suggestions of things to fix. Even if the program runs, you should heed ALL compiler messages, even if they seem miniscule. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Integers (INT)** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Properly cast to enumeration values within range. |

| **Noncompliant Code** |
| --- |
| The noncompliant code checks if a value is within range but does so after casting to an enumeration type, possibly causing the integer value to not be represented properly. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);      if (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| The complaint code checks the value prior to conversion. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    if (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** Architect and design for security policies and practice defense in depth. Knowing that overflow or underflow can lead to potential security vulnerabilities, we need to design programs with as little potential vulnerabilities as possible and ensure there are multiple layers of defense. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Containers (CTR)** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Make sure to iterate over a valid range when iterating over elements in a container. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the first iterator does not precede the second. |
| #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** |
| --- |
| In the compliant code, the iterator values are passed in proper order. |
| #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):** Heed compiler warnings. In the example of the noncompliant code, we would see a definite compiler warning. While this is a simple fix, we always need to heed the compiler. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Characters and Strings (STR)** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Make sure to use valid references, pointers, and iterators when referencing elements of a container. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the iterator is invalidated after the first call to insert(). |
| #include <string>    void f(const std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();    for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| In the compliant code, the iterator is updated as a result of each call to insert(). |
| #include <string>    void f(const std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();    for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

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

| **Principles(s):** Adopt a secure coding standard. This is important for all of the coding standards, but particularly here. Having a secure standard doesn’t just mean ensuring proper defensive measures, but also writing clean and precise code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Memory Management (MEM)** |
| --- | --- | --- |
| **Assertions** | [STD-006-CCP] | When dealing with objects lifetimes, follow the stages to properly construct and deconstruct. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, a class is created but the constructor for the object is never called. |
| #include <cstdlib>    struct S {    S();      void f();  };    void g() {    S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));      s->f();      std::free(s);  } |

| **Compliant Code** |
| --- |
| In the compliant code, the constructor and destructor are explicitly called. |
| #include <cstdlib>  #include <new>    struct S {    S();      void f();  };    void g() {    void \*ptr = std::malloc(sizeof(S));    S \*s = new (ptr) S;      s->f();      s->~S();    std::free(ptr);  } |

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

| **Principles(s):**Architect and design for security policies and adopt a secure coding standard. We always make sure when we create classes, that they are being called properly and used as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Input Output (FIO)** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When opening a file using the open function, there must be a matching close function when the last pointer that stores the return value of the call has ended or program termination. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the object is opened but never 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;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| In the compliant code, the file is opened and closed properly prior to termination. |
| #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();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

| **Principles(s):** Validate input data and sanitize data sent to other systems. When we are importing files or data from other sources, we must always validate and test before we commit to the data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Exceptions and Error Handling (ERR)** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | Properly delete resources when throwing an exception, instead of leaking. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the call is not deleted when an exception is thrown, causing a resource leak. |
| #include <new>    struct SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.    void process\_item() noexcept(false);  };    void f() {    SomeType \*pst = new (std::nothrow) SomeType();    if (!pst) {      // Handle error      return;    }      try {      pst->process\_item();    } catch (...) {      // Process error, but do not recover from it; rethrow.      throw;    }    delete pst;  } |

| **Compliant Code** |
| --- |
| In the compliant code, the handler deletes the call. |
| #include <new>    struct SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.      void process\_item() noexcept(false);  };    void f() {    SomeType \*pst = new (std::nothrow) SomeType();    if (!pst) {      // Handle error      return;    }    try {      pst->process\_item();    } catch (...) {      // Process error, but do not recover from it; rethrow.      delete pst;      throw;    }    delete pst;  } |

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

| **Principles(s):** Architect and design for security policies and adopt a secure coding standard. Resource leaks can lead to vulnerabilities. We need to ensure resources and deleted after use, as to not open vulnerabilities or cause program failure. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Object Oriented Programming (OOP)** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | When writing constructor member initializers, write them in canonical order to alleviate errors. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the initializers are not written in canonical order, creating an unspecified value to be stored. |
| class C {    int dependsOnSomeVal;    int someVal;    public:    C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| In the compliant code, all initializers are canonical. |
| 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. With a secure coding standard comes great coding procedures, such as writing and calling initializers canonically. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Concurrency (CON)** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | When dealing with mutex objects, do not destroy a mutex object while a thread is blocked waiting for a lock. This will leave data unprotected. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, the mutex’s destructor may be called before all threads have been exited, leaving data unprotected. |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    void start\_threads() {    std::thread threads[maxThreads];    std::mutex m;      for (size\_t i = 0; i < maxThreads; ++i) {      threads[i] = std::thread(do\_work, i, &m);    }  } |

| **Compliant Code** |
| --- |
| In the compliant code, the mutex’s lifetime is extended, preventing a premature destructor invoke. |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    std::mutex m;    void start\_threads() {    std::thread threads[maxThreads];      for (size\_t i = 0; i < maxThreads; ++i) {      threads[i] = std::thread(do\_work, i, &m);    }  } |

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

| **Principles(s):** Adopt a secure coding standard, architect and design for security policies, practice defense in depth. We can never, under any circumstance, allow user data to be unprotected. During the design phase, we need to ensure there are multiple layers of defense. We also need to ensure that we are calling and destroying properly, leaving all data protected and access calls deleted after use. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Static analyzer that checks code for compliance with MISRA and CERT guidelines, along with various others. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

There is no need to modify the existing DevOps process, instead, let’s add bits and pieces to pre-existing portions of the process. For instance, under assess and plan, we start the DevOps process by going over this policy in detail, how it applies to the DevOps process and the work being done and what to expect in the upcoming portions of the process. During the later half of the design portion, developers need to go over the design multiple times, doing non-functional testing. During the build portion, developers need to verify they are sticking to the principles and standards. During each phase of the build portion, there also needs to be non-functional and functional testing, looking for bugs, security risks, etc. Verify and test is much of the same as the build portion. Once we work through transition and health check and into monitor and detect, this is where the majority of testing and debugging will happen, along with making sure the final product adheres to the security policy.

### 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 | High | P3 | L3 |
| STD-002-CPP | High | Probable | High | P6 | L2 |
| STD-003-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-004-CPP | High | Probable | High | P6 | L3 |
| STD-005-CPP | High | Probable | High | P6 | L2 |
| STD-006-CPP | High | Likely | Medium | P18 | L1 |
| STD-007-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-008-CPP | Low | Probable | High | P2 | L3 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Medium | Probable | High | P4 | L3 |

### 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 | This type of encryption keeps data secured by keeping said data encrypted when on disk. For instance, even if someone with malicious intent obtains your hardware, they still have to penetrate the encryption to access your private data. |
| Encryption at flight | This type of encryption secures data while the data is being transmitted. |
| Encryption in use | This type of encryption is the future of encryption. Homomorphic encryption allows for a user to use data and change it as they normally would, while encrypting said data at almost the same speed of unencrypted data. Data is never decrypted, even when in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying login credentials against a database to see if the credentials being input are in the database. If they are good credentials, we move on to authorization. |
| Authorization | Authorization verifies what level of access a user has. Mary Sue may have basic user credentials and authorization, while the supervisor, Tammy Sue may have admin credentials and authorization, allowing more access to data. |
| Accounting | Accounting keeps a record of what a user is doing after they have been verified and given access. |

**\***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.

1. Declarations and Initialization

2, 3, 4, 5, 6, 8, 9, 10. With declarations and initializations, we are opening up possible vulnerabilities, if done incorrectly. Each of the principles I chose shows the importance of defense, setting it up for the least amount of damage to be done by a hacker and also how we need to pay attention, not only to our coding abilities but attention to the compiler’s warning.

1. Expressions

2, 4, 10. Without proper coding standards, we risk potential bugs with our program.

1. Integers

2,3, 4, 10. Integers, overflowing and underflowing also lead us to potential vulnerabilities. We need to ensure that this does not happen.

1. Containers

2, 4, 10. Same as with expressions, we run the risk of not having a functioning program without proper coding standards.

1. Characters and Strings

2, 4, 10. Same as with expressions and containers, not having a strong foundation for coding will leave you with bugs or program that doesn’t work.

1. Memory Management

2, 3, 5, 6, 8, 9, 10. When it comes to memory, we also see potential vulnerabilities with overflows and underflows. Memory needs multiple layers of defense, a proper architecture, and strong code to ensure it runs as efficiently and safely as possible.

1. Input Output

1, 7, 10. When importing files or data from untrusted sources, they need to be thoroughly tested and validated. This is a vulnerability waiting to happen.

1. Exceptions and Error Handling

2, 4, 9, 10. When writing exceptions and errors to be thrown under certain circumstances, we need to have a good coding standard to ensure they work, are easily understandable, and do not false flag.

1. Object Oriented Programming

2, 3, 9, 10. As with the rest of development, trust your compiler and heed the warnings. Even if the program runs, if there are any issues coming up in the compiler, fix it.

1. Concurrency

2, 3, 8, 9, 10.

We must follow Defense in Depth and have a strong coding standard to ensure data is not left unprotected. When user data is brought into the mix, strong architecture is foundation for many layers of even stronger defense, protecting the user data.

**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.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [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 |