**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 | Always check data from sources you don't completely trust. Properly validating inputs can fix most software problems and keep your system safe. Be cautious with external data sources like network connections, command line inputs, environment settings, and user-controlled files. By carefully examining these sources, you can greatly reduce the risk of security issues. |
| 1. Heed Compiler Warnings | Compile your code with the highest warning level your compiler supports and address any warnings by adjusting the code. Use both static and dynamic analysis tools to identify and correct additional security vulnerabilities. |
| 1. Architect and Design for Security Policies | To set and enforce security policies, design a software architecture for your program. If your system requires different levels of access at various times, consider dividing it into separate, interacting subsystems, each with its own specific privileges. |
| 1. Keep It Simple | Aim for a design that is small and easy to understand. The more complicated your design is, the more likely it is to have mistakes during its setup, configuration, and use. Also, as security methods get more complex, it requires more effort to make sure they are effective and reliable. Keeping things simple helps reduce the chances of errors and makes it easier to ensure that everything works as it should. |
| 1. Default Deny | Base access decisions on permissions rather than on exclusions. This approach means the security system defines the conditions under which access is granted, and by default, access is denied. |
| 1. Adhere to the Principle of Least Privilege | All processes should operate with only the essential privileges needed to complete their tasks. Any elevated permissions should be kept for the shortest time possible. This approach reduces the risk of an attacker being able to execute unauthorized code with higher privileges. |
| 1. Sanitize Data Sent to Other Systems | Ensure that any data sent to complex subsystems such as relational databases, command shells, and third-party components is properly sanitized. Attackers might exploit SQL, command, or other injection methods to access unused functions within these components. Since these complex subsystems do not understand the context of the input, it isn't always an issue of input validation. The responsibility for cleaning the data before interacting with the subsystem lies with the calling process, as it knows the context of the data. |
| 1. Practice Defense in Depth | Use a range of protective measures to manage risk. This way, if one defense fails, another can stop a security problem from becoming a serious issue or reduce the damage if an exploit happens. For example, combining safe runtime environments with good coding practices can help prevent any remaining vulnerabilities from being exploited once the system is in use. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance methods can help find and fix vulnerabilities. A strong quality assurance program should include checking the source code, performing penetration tests, and running fuzz tests. Systems that are reviewed by independent security experts are often more secure. These outside reviewers offer an unbiased view that can help spot and correct incorrect assumptions. |
| 1. Adopt a Secure Coding Standard | Develop and adopt a secure coding standard tailored to the programming language and platform you are using. This standard should provide detailed guidelines and best practices for writing code securely. It ensures that all code written within your chosen environment adheres to principles designed to protect against common vulnerabilities and security threats. By having a well-defined secure coding standard, you can help prevent security issues and maintain the integrity of your software throughout its development lifecycle. |

### 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-nnn-LLL] | Obey the one definition rule. |

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, two separate translation units each define a class with the same name but with different definitions. Even though both definitions are functionally equivalent (each defines a class named S with a single public, non-static integer member named `a`), the sequences of tokens used in the definitions differ. This discrepancy violates the One Definition Rule (ODR) and leads to undefined behavior. |
| // a.cpp  **struct** S {  **int** a;  };    // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| The right way to address the issue depends on the programmer's goal. If the intention is for the same class definition to be available in both translation units due to its shared use, the solution is to place the class definition in a header file and include that header file in both translation units, as shown in this compliant example. |
| // 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):**  Architect and Design for Security Policies  Keep It Simples  Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Type-compatibility  Definition-duplicate  Undefined-extern  Undefined-extern-pure-virtual  External-file-spreading  Type-file-spreading | Partially checked |
| CodeSonar | 8.0p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| LDRA Tool Suite | 9.7.1 | 286 S, 287 S | Fully implemented |
| Parasoft C/C++test | 2023.1 | CERT CPP-DCL60-a | A class, union or enum name (including qualification, if any) shall be a unique identifier |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-nnn-LLL] | Do not read uninitialized memory. |

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, an uninitialized local variable is used in an expression to print its value, leading to undefined behavior. |
| #include <iostream>    **void** f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized before its value is printed. |
| #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  Keep It Simple  Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Uninitialized read | Partially checked |
| Clang | 3.9 | Wuninitialized  Clang-analyzer-  Core.unidefinedbinaryoperatorresult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| CodeSonar | 8.0P0 | 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-nnn-LLL] | Do not attempt to create a std::string from a null pointer. |

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, a `std::string` object is constructed using the results of a call to `std::getenv()`. However, since `std::getenv()` returns a null pointer if it fails, this code can result in undefined behavior if the environment variable is not present or if another error occurs. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to `std::getenv()` are verified for null before creating the `std::string` object. |
| #include <cstdlib>  #include <string>    void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** Heed Compiler Warnings |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Assert Failure |  |
| CodeSonar | 8.0p0 | Lang.MEM.NPD | Null pointer deference |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023b | CERT C++: STR51-CPP | This rule partially checks for string operations on null pointers, meaning it looks for errors when strings are used with null pointers. However, it doesn’t cover every possible case, so some issues with null pointers and string operations might still be missed. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-nnn-LLL] | Do not store an already-owned pointer value in an unrelated smart pointer. |

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, two separate smart pointers are created from the same underlying pointer value. When the local, automatic variable `p2` is destroyed, it deletes the pointer it manages. Later, when the local, automatic variable `p1` is destroyed, it attempts to delete the same pointer, leading to a double-free vulnerability. |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the `std::shared\_ptr` objects are linked through copy construction. When the local, automatic variable `p2` is destroyed, the reference count for the shared pointer is decreased but remains greater than zero. Later, when the local, automatic variable `p1` is destroyed, the reference count drops to zero, causing the managed pointer to be deleted. Additionally, this solution uses `std::make\_shared()` to allocate memory and manage the pointer, rather than manually creating a raw pointer and storing it in a local variable. |
| #include <memory>    void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):**  Validate Input Data  Sanitize Data Sent to Other Systems  Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Dangling\_pointer\_use |  |
| Helix QAC | 2023.3 | DF4721, DF44722, DF4723 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM56-a | Avoid storing a pointer that is already owned by another smart pointer in an unrelated smart pointer. |
| Polyspace Bug Finder | R2023b | CERT C++: MEM56-CPP | Verifies the use of pointers that are already owned (fully covers the rule). |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-nnn-LLL] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, `s` is dereferenced after its memory has been deallocated. If this dereference leads to a write-after-free situation, it could be exploited to execute arbitrary code with the permissions of the vulnerable process. Because dynamic memory allocations and deallocations often occur in different parts of the code, it can be challenging to identify and diagnose these issues. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is kept allocated until it is no longer needed. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  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):**  Heed Compiler Warnings  Default Deny  Adhere to the Principle of Least Privilege  Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus. NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Verified by clang-tidy but doesn't detect all rule violations. |
| CodeSonar | 8.0p0 | ALLOC.UAF | Use only after free |
| Coverity | V7.5.0 | USE\_AFTER\_FREE | Can identify cases where memory is freed multiple times or where a freed pointer is accessed for reading or writing. |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the `assert()` macro to check a condition related to a memory-mapped structure that is crucial for the code to function correctly. |
| #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** |
| --- |
| For assertions that involve only constant expressions, a preprocessor conditional statement can be used, as demonstrated in this compliant solution. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

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

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Verified by clang-tidy |
| 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-nnn-LLL] | Handle all exceptions thrown before main() begins executing. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the constructor for `S` might throw an exception that goes uncaught when `globalS` is initialized during the program's startup. |
| struct S {  S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| This compliant solution changes `globalS` into a local variable with static storage duration, ensuring that any exceptions thrown during its construction can be caught. The constructor for `S` is executed only when the `globalS()` function is first called, rather than at program startup. This approach does require modifying the source code so that previous instances of `globalS` are replaced with a call to `globalS()`. |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques  Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Potentially-throwing-static-initialization | Partially checked |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| CodeSonar | 8.0p0 | LANG.STRUCT.EXCP.THROW | Use of throw |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR58-a | Exceptions should only be raised after the program has started and before it terminates. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-nnn-LLL] | Use valid iterator rangesthe standard. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the two iterators defining the range point to the same container, but the first iterator does not come before the second. Within its internal loop, `std::for\_each()` increments the first iterator and then compares it with the second iterator for equality. As long as they are not equal, the function keeps incrementing the first iterator. Incrementing an iterator that represents the end of the range leads to 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** |
| --- |
| In this compliant solution, the iterator values provided to `std::for\_each()` are supplied in the correct 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):**  Architect and Design for Security Policies  Keep It Simple  Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Overflow\_upon\_dereference |  |
| CodeSonar | 8.0p0 | LNG.MEM.BO | Overrun of buffer |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53b | Avoid using an iterator range that does not represent a valid range. Do not compare iterators from separate containers. |
| [Insert text.] | [Insert text.] | [Insert text.] | Checks for incorrect iterator ranges (rule partially enforced). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-nnn-LLL] | Write constructor member initializers in the canonical order |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the member initializer list for `C::C()` tries to initialize `someVal` first, followed by initializing `dependsOnSomeVal` with a value that depends on `someVal`. Since the order of the member variable declarations does not match the order in the initializer list, reading the value of `someVal` may result in an undefined value being assigned to `dependsOnSomeVal`. |
| class C {  int dependsOnSomeVal;  int someVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| This compliant solution adjusts the declaration order of the class member variables so that the dependencies are properly sequenced in the constructor's member initializer list. |
| 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):** Keep It Simple |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Initializer-list-order | Fully checked |
| CodeSonar | 8.0p0 | LANG.STRUCT.INIT.OOMI | Incorrect Order of Member Initializers |
| LDRA tool suite | 9.7.1 | 206 S | This is fully implemented. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-OOP54-a | Initialize members in the same order as they are declared. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-nnn-LLL] | Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object before the pointer's lifetime has begun, leading to undefined behavior. |
| struct S {  void mem\_fn();  };    void f() {  S \*s;  s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, memory is allocated for the pointer before calling `S::mem\_fn()`. |
| 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  Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Return-reference-local dangling\_pointer\_use | This is partially checked |
| Clang | 3.9 | Wdangling-initializer-list | This catches some lifetime issues that are related to the incorrect use of std: :initializer\_list<> |
| CodeSonar | 8.0p0 | IO.UAC  ALLOC.UAF | Use it after close  Use it after free |
| LDRA tool suite | 2023.1 | 42 D, 53 D, 77 D, 1 J, 71 S, 565 S | This is partially implemented. |

### 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.

To implement a DevSecOps framework successfully, it is important to start by reviewing your existing DevOps policies. DevOps is known for its efficiency in releasing new software, but it often overlooks critical security concerns. DevSecOps addresses this by making security a fundamental part of the process right from the beginning. It treats security as equally important as software development and operations, aiming to change the engineering culture to focus on security. The main goal of DevSecOps is to apply security practices throughout every phase of the Software Development Lifecycle (SDLC).

To align our process with DevSecOps standards, I would make several changes to each part of our traditional DevOps lifecycle. First, during the planning phase, it is crucial to consider common security threats that could impact the software. For example, threats like SQL Injection or Man-in-the-Middle attacks are well-known risks. I would design our software with defenses against these threats from the very start. This means planning for security measures and building them into our project plans.

In the development and build stages, it is important to integrate security practices and guidelines into our workflow. This includes incorporating secure coding practices to minimize instability and reduce the risk of external manipulation. By embedding security into our development process, we make sure that potential vulnerabilities are addressed early, and that the software is less likely to be compromised by external attacks.

During the testing phase, I would use automated unit tests to check each small component of the software. These tests help ensure that each part works correctly and securely. Additionally, I would conduct integration testing to examine the entire application stack. This type of testing looks at the whole system, from the user interface to the database, and helps identify any weaknesses or security gaps.

For the release, deployment, operation, and monitoring stages, I would implement a secure container system. This system helps protect the underlying operating system from unauthorized access, ensuring that only authorized users can interact with the software. Automated log analysis would be used to detect potential threats before they cause system failures. By analyzing logs automatically, we can find and fix issues quickly. Additionally, monitoring network traffic is essential to identify unusual patterns or high volumes of traffic that could indicate a denial-of-service (DDOS) attack or other security problems. By keeping a close watch on network activity, we can respond to potential threats promptly and keep the system secure.

In summary, applying DevSecOps principles means making security a priority throughout the entire software development process. By incorporating security into each stage, from planning to deployment and monitoring, we ensure that our software is well-protected against potential threats and vulnerabilities.

### 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 | Low | 3 |
| STD-002-CPP | High | Probable | Medium | High | 1 |
| STD-003-CCP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | High | Low | 3 |
| STD-007-CPP | Low | likely | Low | Medium | 2 |
| STD-008-CPP | High | Probable | High | Medium | 2 |
| STD-009-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-010-CPP | High | Probable | High | Medium | 2 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Data stored on various devices and platforms is safeguarded through encryption. This includes information on hard drives, smartphones, personal computers, and cloud storage. To secure sensitive data, encryption methods such as disk encryption and security tools for computers and mobile devices are employed. |
| Encryption in flight | Encryption during transmission aims to protect data as it moves, whether it's traveling outside a network or between devices within the same network. To ensure this protection, methods like email encryption, Data Loss Prevention (DLP) programs, and robust network security measures such as firewalls and authentication are utilized. It's also crucial to consider both the path the data takes and the security of that path. |
| Encryption in use | Data that is actively being used, such as during creation or updates, is safeguarded through encryption. To ensure its security, it's important to have measures in place for controlling and protecting the data both before and after its use. Managing identity and access rights effectively can also help minimize the risk to this data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of confirming a person's identity. It can involve various methods, such as biometric data, certificates, static passwords, and one-time passwords. These verification techniques ensure that an individual is accurately identified. |
| Authorization | Authorization is crucial in computer and information security as it determines a user's access rights after their identity is verified through authentication. While authentication confirms who a user is, via passwords, biometrics, or security tokens, authorization defines what they can access and do within the system. For instance, an employee might access general documents, while a manager might access sensitive financial reports. Authorization limits security risks by ensuring users only access information and perform actions relevant to their role, thus protecting sensitive data from misuse or unauthorized access. |
| Accounting | Accounting involves tracking system activities by recording data transfers, resource usage, and timestamps. This process creates a detailed log of user actions, which is useful for understanding user behavior and, if necessary, conducting forensic investigations to analyze any suspicious activity. |

**\***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 |  |
| 1.1 | 7/28/2024 | Module 3 | Ruben Sanchez |  |
| 1.2 | 8/18/2024 | Module 6-Project 1 | Ruben Sanchez |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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