**Green Pace Developer: Security Policy Guide Template**

**Jacob Casas**

**CS-405**



# 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 | Check all user input carefully for potentially dangerous code, unusual forms, and injection threats before processing it. Consider every input to be unreliable and thoroughly verify it. |
| 1. Heed Compiler Warnings | Compiler warnings about security flaws like buffer overflows and memory leaks should not be disregarded. Quickly address them to stop attackers from taking use of these vulnerabilities. |
| 1. Architect and Design for Security Policies | Include security concerns in the design process. Reduce attack surfaces and encourage secure defaults by clearly defining security policies and making sure every component abides by them. |
| 1. Keep It Simple | Securing complex code is more difficult. To minimize possible vulnerabilities and improve code maintainability, embrace simplicity, modularity, and well-defined functionality. This will promote a better knowledge of security. |
| 1. Default Deny | Put into practice a "deny-by-default" method for access management. Permissions should only be granted when absolutely essential and should be revoked once no longer needed. In the event that an attacker compromises the system, this reduces the possible harm. |
| 1. Adhere to the Principle of Least Privilege | Provide users and procedures the minimal amount of access required to complete their responsibilities. By doing this, the effects of compromised accounts or vulnerabilities are lessened, and attackers are prevented from gaining more authority. |
| 1. Sanitize Data Sent to Other Systems | Never put all your faith on external systems. Before sending any data to other systems, clean it up to get rid of any potentially dangerous components and stop injection attacks. |
| 1. Practice Defense in Depth | Use more than one layer of security measures; don't stop at one. since of this redundancy, it is more difficult for attackers to fully compromise the system since other levels can take over in the event that one fails. |
| 1. Use Effective Quality Assurance Techniques | Don't stop at the conclusion of the development process; continue doing security testing all the way through. To find and fix vulnerabilities early on, use penetration testing, static code analysis, and other pertinent methods. |
| 1. Adopt a Secure Coding Standard | Create and implement a clear coding standard that prioritizes safe coding techniques. This encourages uniformity, lessens frequent errors, and guarantees that developers give security first priority while writing. |

### 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** | **INT50-CPP** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | There are two types of enumerations in C++: unscoped enumerations, where the underlying type is not fixed, and scoped enumerations, where the underlying type is fixed. Both types of enumerations can represent a range of values, including enumerator values that aren't stated in the enumeration itself. The C++ Standard, [dcl.enum], in paragraph 8, defines the range of acceptable enumeration values for an enumeration type. |

| **Noncompliant Code** |
| --- |
| The goal of this noncompliant code sample is to determine if a given number falls into the permitted range of enumeration values. It may not be able to represent the supplied integer value, though, because it is casting to the enumeration type first. If a value outside of that range were supplied to f(), the cast to EnumType would result in an ambiguous value, and utilizing that value within the if statement would result in unspecified behavior. On a two's complement system, the valid range of values that may be represented by EnumType are [0..3]. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| In order to ensure that the conversion does not produce an undefined value, this conforming solution first verifies that the value can be represented by the enumeration type. This is accomplished by limiting the converted value to those for whom an enumerator value exists. |
| **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):** CERT-CPP-INT50-A This principle specifies that "An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration." To put it another way, if an integer value doesn't match one of the stated enumerators, you should never convert it to an enumeration type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | cast-integer-to-enum | Partially checked |
| CodeSonar | 8.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| Polyspace Bug Finder | R2023b | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **CTR51-CPP** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | A C++ program may operate uniformly with various data structures (containers) by using iterators, which are a generalization of pointers [ISO/IEC 14882-2014]. There is a close link between pointers, references, and iterators. Using a correct iterator, pointer, or reference is necessary when referring values. There is a chance that if you hold an iterator, reference, or pointer to an element inside of a container for an extended period of time, the underlying container might change and the stored iterator, pointer, or reference would become invalid. For example, outstanding iterators, pointers, and references will be invalidated when a sequence container like std::vector needs an underlying reallocation [Kalev 99]. Refer to an element of a container by only using a valid pointer, reference, or iterator. |

| **Noncompliant Code** |
| --- |
| This noncompliant code sample has loop iterations with unclear behavior after the first insert() call, and pos is invalidated after that. |
| #include <deque>    **void** f(**const** **double** \*items, std::**size\_t** count) {    std::deque<**double**> d;    auto pos = d.begin();  **for** (std::**size\_t** i = 0; i < count; ++i, ++pos) {      d.insert(pos, items[i] + 41.0);    }  } |

| **Compliant Code** |
| --- |
| To avoid undefinable behavior, this conforming approach assigns pos a valid iterator at each insertion. |
| #include <deque>    **void** f(**const** **double** \*items, std::**size\_t** count) {    std::deque<**double**> d;    auto pos = d.begin();  **for** (std::**size\_t** i = 0; i < count; ++i, ++pos) {      pos = d.insert(pos, items[i] + 41.0);    }  } |

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

| **Principles(s):** CERT-CPP-CTR51-A It is stated in this principle: "Do not modify a container while iterating over it." Stated otherwise, you shouldn't use iterators, pointers, or references to iterate through a container's elements while also changing the container's elements or structure. This is due to the possibility that these changes might render the iterators invalid, resulting in unpredictable behavior and possible security flaws. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | ALLOC.UAF | Use After Free |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| Polyspace Bug Finder | R2023b | CERT C++:CTR51-CPP | Checks for use of invalid iterator (rule partially covered) |
| Astree | 22.10 | Overflow\_upon\_dereference |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **STR53-CPP** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | These index operations for std::string operator[]const\_reference(size\_type) The character stored at the given location, pos, is returned by the const and reference operators[](size\_type). A reference to an object of type charT with value charT() is returned when pos >= size(). There are no exceptions thrown for range problems, therefore the index operators are unchecked. Moreover, modifying the resultant out-of-range object leads to undefined behavior. |

| **Noncompliant Code** |
| --- |
| The value provided by the get\_index() function in this noncompliant code sample could be more than the number of items kept in the string, which would lead to undefined behavior. |
| **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");    s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| The std::basic\_string::at() method, which functions similarly to the index operator[] but raises a std::out\_of\_range exception if pos >= size(), is used in this complying approach. |
| #include <stdexcept>  #include <string>  **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");  **try** {      s.at(get\_index()) = '1';    } **catch** (std::out\_of\_range &) {      // Handle error    }  } |

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

| **Principles(s):** CERT-CPP-STR53-A "Ensure that container indices are within the valid range," according to this principle. Stated otherwise, you have to be sure that the indices you use to access elements of a container (such as std::string) do not exceed the container's boundaries. When components are accessed beyond of their valid range, unpredictable behavior results, sometimes with dire security ramifications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type Overrun  Type underrun |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2023b | CERT C++: STR53-CPP | Checks for:   * Array Access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered |
| Astree | 22.10 | assert\_failure |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **MEM56-CPP** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | As part of the type system, smart pointers like std::unique\_ptr and std::shared\_ptr encode pointer ownership semantics. They govern the lifetime of the pointer they manage, encapsulate a pointer value, and offer pointer-like semantics through member operations like operator \*() and operator ->(). A value is said to be owned by the smart pointer when it is created from a pointer value. |

| **Noncompliant Code** |
| --- |
| Two unrelated smart pointers are built from the same underlying pointer value in this example of noncompliant programming. The pointer value it controls is deleted with the destruction of the local, automated variable p2. The identical pointer value is then deleted with the destruction of the local, automated variable p1, creating 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** |
| --- |
| The std::shared\_ptr objects in this compatible approach are coupled to each other via copy construction. Upon the destruction of the local, automatic variable p2, the shared pointer value's use count decreases, but it remains nonzero. The managed pointer is then removed together with the local, automated variable p1 when the shared pointer value's use count is reduced to zero. Additionally, rather to creating a raw reference and storing its value in a local variable, this compatible approach utilizes std::make\_shared(). |
| #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):** CERT-CPP-MEM56-A This principle says, "Do not store an already-owned pointer value in an unrelated smart pointer." To put it another way, do not create multiple smart pointers that share the same raw pointer memory. When memory is deallocated twice, it can result in double-free vulnerabilities that can lead to application failures, corrupted data, and even security exploits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| Polyspace Bug Finder | R2023b | CERT C++:MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |
| Astree | 22.10 | Dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM56 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **MEM51-CPP** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | A C++ application can utilize the memory allocation functions std::malloc(), std::calloc(), and std::realloc(), which are provided by the C programming language. Nevertheless, std::free() is the sole method defined by the C programming language for freeing up allocated memory. Refer to MEM31-C. When not in use, release dynamically allocated memory and MEM34-C. The only dynamically allocated free memory is for rules that expressly address deallocation needs and C allocation. |

| **Noncompliant Code** |
| --- |
| The local variable space is given as the expression to the placement new operator in this example of noncompliant code. After that call, the resultant reference is handed to ::operator delete(), which attempts to clear memory that ::operator new() did not return, leading to undefined behavior. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...    **delete** s1;  } |

| **Compliant Code** |
| --- |
| This conforming approach calls s1's destructor directly rather to removing the call to ::operator destroy(). There are very few situations in which calling a destructor explicitly is appropriate. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...      s1->~S();  } |

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

| **Principles(s):** CERT-CPP-MEM51-A, CERT-CPP-MEM51-B, and CERT-CPP-MEM51-D Together, these principles cover how to use C++'s operator new and operator delete to manage memory effectively. |
| --- |

**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.NewDeleteLeaks  -Wmismatched-new-delete  Clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 8.0p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470S, 483 S, 484 S, 485 S, 64D, 112 D | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets ([]) for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resouces |
| Polyspace Bug Finder | R2023b | CERT C++:MEM51-CPP | Checks for:   * Invalid deletion of pointer * Invalid free of pointer * Deallocation of previously deallocated pointer   Rule partially covered |

#### Coding Standard 6

| **Coding Standard** | **Label** | **EXP20-C** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Conduct explicit tests to ascertain equality, success, and true/false in order to enhance code readability and maintainability and to ensure compliance with standard practices. Specifically, avoid setting the test for nonzero by default. Let's say the foo() method returns a nonzero result in the case of success and 0 otherwise, indicating failure. Using 0 to test for inequality |

| **Noncompliant Code** |
| --- |
| Is\_banned() in this noncompliant code sample returns nonzero in the case of yes and 0 otherwise: |
| LinkedList bannedUsers;    **int** is\_banned(User usr) {  **int** x = 0;      Node cur\_node = (bannedUsers->head);    **while** (cur\_node != NULL) {  **if**(!**strcmp**((**char** \*)cur\_node->data, usr->name)) {        x++;      }      cur\_node = cur\_node->next;    }    **return** x;  }    **void** processRequest(User usr) {  **if**(is\_banned(usr) == 1) {  **return**;    }    serveResults();  } |

| **Compliant Code** |
| --- |
| Because most functions only guarantee a nonzero return value for true, it would be preferable to rewrite the same code as follows to check for inequality with 0 (false): |
| LinkedList bannedUsers;    **int** is\_banned(User usr) {  **int** x = 0;      Node cur\_node = (bannedUsers->head);    **while**(cur\_node != NULL) {  **if** (**strcmp**((**char** \*)cur\_node->data, usr->name)==0) {        x++;      }      cur\_node = cur\_node->next;    }    **return** x;  }    **void** processRequest(User usr) {  **if** (is\_banned(usr) != 0) {  **return**;    }    serveResults();  } |

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

| **Principles(s):** CERT-C-EXP20-A and CERT-C-EXP20-B Together, these principles encourage precise and unambiguous assertions in C code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported indirectly via MISRA C:2004 Rule 13.2 |
| Axivion Bauhaus Suite | 7.2.0 | CERTC-EXP20 | Fully Implemented |
| LDRA tool suite | 9.7.1 | 114 S | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_C-EXP20-a  CERT\_C-EXP20-b | Avoid comparing values with TRUE macro/enum constant using equality operators (“==”, “!=”)  Tests of a value against zero should be made explicit, unless the operand is effectively Boolean |
| PC-lint Plus | 1.4 | 697 | Partially supported: reports comparisons of Boolean values to constants other than 0 |

#### Coding Standard 7

| **Coding Standard** | **Label** | **ERR57** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | It's crucial to recover resources when exceptions are raised. An object might be left in a partly initialized state or cleanup code could be skipped as a result of throwing an error. According to ERR56-CPP, such a partly initialized object would go against fundamental exception safety. Ensure exceptional security. When objects fall out of scope, it is ideal that resources be automatically recovered using the RAII design pattern [Stroustrup 2001]. This method save you from having to write a lot of cleaning code when assigning resources. |

| **Noncompliant Code** |
| --- |
| When process\_item raises an exception in this noncompliant code sample, pst is not appropriately relinquished, leading to 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** |
| --- |
| Using the delete function, the exception handler in this conforming solution releases PST. |
| #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):** CERT-CPP-ERR57-A In accordance with this principle, "Ensure resources are freed when exceptions are raised." Basically, in order to avoid leaks and preserve program integrity, it is imperative to ensure that any related resources are appropriately relinquished when an error arises during object usage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | ALLOC.LEAK | Leak |
| LDRA tool suite | 9.7.1 | 50 D | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR57-a | Ensure resources are freed |
| Polyspace Bug Finder | R2023b | CERT C++:ERR57-CPP | Checks for:   * Resource leak caused by exception * Object left in partially initialized state * Bad allocation in constructor |

#### Coding Standard 8

| **Coding Standard** | **Label** | **CTR52-CPP** |
| --- | --- | --- |
| Containers | [STD-008-CPP] | A buffer overflow occurs when data is copied into a container that is not big enough to accommodate that data. Data copied to the destination container must be limited based on its size in order to avoid such problems; alternatively, the destination container must be ensured to be big enough to accommodate the data that has to be copied. |

| **Noncompliant Code** |
| --- |
| Array data types and STL containers are susceptible to similar issues. The std::copy() method can cause a buffer overflow and lacks built-in bounds checking. This noncompliant code sample uses std::copy() to copy a vector of integers from source to dest. When copying the initial element, the program will overrun the buffer since std::copy() does not enlarge the dest vector. |
| #include <algorithm>  #include <vector>    **void** f(**const** std::vector<**int**> &src) {    std::vector<**int**> dest;    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

| **Compliant Code** |
| --- |
| Making sure the destination container has enough space to accommodate all of the pieces being transferred to it is the correct method to utilize std::copy(). The vector's capacity is increased by this conforming solution before the copy process. |
| #include <algorithm>  #include <vector>  **void** f(**const** std::vector<**int**> &src) {    // Initialize dest with src.size() default-inserted elements    std::vector<**int**> dest(src.size());    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

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

| **Principles(s):** CERT-CPP-CTR52-A "Do not pass empty container iterators to std algorithms as destinations," states this principle. Put another way, make sure the destination container has enough space to keep the duplicated pieces when utilizing container-operating techniques like std::copy. If not, buffer overflows might occur, causing memory corruption and possibly even security flaws. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | BADFUNC.BO.\*  LANG.MEM.BO  LANG.MEM.TBA | A collection of warning classes that report uses of library functions prone to internal buffer overflows.  Buffer Overrun  Tainted Buffer Access |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR52-a | Do not pass empty container iterators to std algorithms as destinations |
| Polyspace Bug Finder | R2023b | CERT C++:CTR52-CPP | Checks for library functions overflowing sequence container (rule partially covered). |
| Astree | 22.10 | invalid\_pointer\_dereference |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **FIO51** |
| --- | --- | --- |
| Input Output | [STD-009-CPP] | When a call is made to std::basic\_filebuf<T>::open(), it must be followed by a call to std::basic\_filebuf<T>::close(), whichever happens first, before the lifetime of the last pointer holding the call's return value expires. |

| **Noncompliant Code** |
| --- |
| This example of noncompliant code creates an object file called std::fstream. Std::basic\_filebuf<T>::open() is called by the constructor for std::fstream, and std::abort(), the default std::terminate\_handler called by std::terminate(), does not call destructors. As a result, the object is not closing the underlying std::basic\_filebuf<T> object correctly. |
| #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 this conforming method, the file resources are closed correctly since std::fstream::close() is called before to std::terminate(). |
| #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):** CERT-CPP-FIO51-A It is stated in this principle, "Ensure resources are freed when opening files." It highlights how crucial it is to correctly close file streams at the end of your work in order to prevent resource leaks and possible file corruption problems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2023b | CERT C++:FIO51-CPP | Checks for resource leak (rule partially covered) |
| Helix QAC | 2023.3 | DF4786, DF4787, DF4788 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **CON50-CPP** |
| --- | --- | --- |
| Concurrency | [STD-010-CPP] | To prevent concurrent access to shared data, mutex objects are employed. Shared data and important parts are unprotected if a mutex object is deleted while a thread is stalled awaiting the lock. |

| **Noncompliant Code** |
| --- |
| This example of noncompliant code generates many threads, each of which calls the do\_work() method with an ID of a distinct integer. Regretfully, because start\_threads() could call the mutex's destructor before every thread has finished, this code has a race situation that allows the mutex to be destroyed while it is still possessed. |
| #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** |
| --- |
| The race problem is removed by this conforming approach by prolonging the mutex's lifespan. |
| #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):** CERT-CPP-CON50-A In accordance with this principle, "Do not destroy another thread's mutex." It highlights how crucial it is to make sure mutexes—which are used to synchronize across threads—are not removed when there may still be threads that depend on them. Trying to lock a mutex that has been destroyed may result in crashes, erratic behavior, or distorted data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | CONCURRENCY.LOCALARG | Local Variable Passed to Thread |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CON50-a | Do not destroy another thread’s mutex |
| Polyspace Bug Finder | R2023b | CERT C++:CON50-CPP | Checks for destruction of locked mutex (rule partially covered) |
| Klocwork | 2023.3 | CERT.CONC.MUTEX.DESTROY\_WHILE\_LOCKED |  |

### 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.

Use technologies such as STRIDE or PASTA during threat modeling to find any weaknesses associated with your security policy requirements. Prioritize security requirements in the backlog according to the seriousness and probability of risks that have been discovered. Incorporate the security policy guidelines into the DevOps process's current risk management frameworks.

During the build process, use Static Application Security Testing (SAST) tools to enforce safe coding principles. These tools may find and highlight code infractions in the development environment that go against the security policies. Utilize infrastructure as a service (IaC) technologies such as Ansible or Terraform to deploy and configure infrastructure in a manner compliant with the security policy. Within IaC templates, provide infrastructure settings that adhere to the security criteria. Before deploying, integrate Infrastructure as Code (IaC) scanning technologies to find and fix security misconfigurations in the infrastructure.

Test security automatically at every stage of the CI/CD process. Employ techniques like fuzzing tools and dynamic application security testing (DAST) to find runtime vulnerabilities that might evade static analysis. To ensure that security policy guidelines are being followed, incorporate automated checks into the CI/CD pipeline. Before deployment, these checks can examine infrastructure, code, and settings for compliance.

When deploying systems, use configuration management tools such as Puppet or Chef to enforce desirable security parameters. Utilize Security Information and Event Management (SIEM) technologies to gather and examine application and infrastructure security logs. Set up SIEM to produce warnings when possible runtime security policy breaches are found.

### 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | High | Probable | High | P6 | L2 |
| STD-003-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Medium | Probable | Low | P12 | L1 |
| STD-007-CPP | Low | Probable | High | P2 | L3 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| 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 | Every important piece of information, whether it's stored on disk, in databases, or in a backup for security, is encrypted using strong algorithms like AES-256. Encryption keys are carefully maintained and kept separate, and backups follow the same strict encryption guidelines. This guarantees that private information won't be accessible even in the event that storage devices are hacked. |
| Encryption at flight | Sensitive data will be protected using robust encryption or industry-standard protocols like TLS/SSL as it travels across internal or external networks. Extra security for really sensitive data will be used, such as VPNs or specific encrypted routes. An additional degree of protection is ensured by securely managing encryption keys apart from the encrypted data. |
| Encryption in use | Sensitive data is encrypted whenever feasible during processing or access, reducing exposure even in temporary files and RAM. Furthermore, whether shared or shown within apps, sensitive information is further obscured via data masking or tokenization techniques. Even in the event of possible system breaches, the danger of exposure is greatly decreased by this multi-layered method. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication uses rigorous techniques like multi-factor authentication and strict password regulations to confirm the identity of individuals and systems trying to access data. Users are only given access rights that they genuinely require, thanks to the concept of least privilege. This barrier is further strengthened by routine audits and inspections of user access. |
| Authorization | Clear guidelines governing who may access what are defined and upheld by authorization. Access control lists (ACLs) are used in conjunction with role-based access control (RBAC) and attribute-based access control (ABAC) to provide granular control over sensitive data and resources. The least privilege concept is still in place, and frequent audits and evaluations of access limits ensure that they stay effective. |
| Accounting | All attempts to access or use private information are tracked and recorded, giving forensic analysts and security breach investigators access to important audit data. Due to their tamper-proof nature and set retention time, logs guarantee accountability and provide prompt action in the event of suspicious behavior. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 02/04/2024 | Partial Template | Jacob Casas |  |
| 3.0 | 02/18/2024 | Completed Template | Jacob Casas |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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