**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 | Validate input from all untrusted sources. Proper input validation can prevent software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user-controlled files |
| 1. Heed Compiler Warnings | When compiling code use the highest warning level available and eliminate warnings by modifying code. Test your code, use static and dynamic analysis tools to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Use appropriate privilege sets, create a software architecture, and design your software to implement and enforce security policy. Only grant access to those who need it to get their work done. |
| 1. Keep It Simple | To reduce errors in implementation, configuration and use keep the design and size simple. The more complex the more your code has chances of error. |
| 1. Default Deny | Deny access to everyone, only grant permission to those who need permission to get their job done. Access should only be given according to the job being done. |
| 1. Adhere to the Principle of Least Privilege | Access should be granted with the least amount of access needed to complete the job. Elevated permission should be granted for the least amount of time needed to complete job. |
| 1. Sanitize Data Sent to Other Systems | Remove any unsafe characters from user inputs and validate expected format. |
| 1. Practice Defense in Depth | Use multiple defense strategies, you must have more than one layer of defense. If one defense is compromised, you should have another layer ready to fight the attack. |
| 1. Use Effective Quality Assurance Techniques | Your code must be fully tested before deployment. Testing is an important step to prevent flaws from becoming exploited. You should use multiple forms of testing for effective quality assurance. |
| 1. Adopt a Secure Coding Standard | For you to develop a successful product you must have code Standards in place for everyone in the company. For successful development everyone should have the same standard of coding. |

### 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** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP]  DCL52-CPP | C++ does not allow you to change the value of a reference type, effectively treating all references as being const qualified. C++ prohibits or ignores the cv-qualification of a reference type. Only a value of non-reference type may be cv-qualified. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example correctly declares p to be a reference to a const-qualified char. The subsequent modification of p makes the program ill-formed. |
| #include <iostream>    void f(char c) {  const char &p = c;  p = 'p'; // Error: read-only variable is not assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the const qualifier. |
| #include <iostream>    void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):** The principle is const-correctness. A function or method should indicate which of its parameters are intended to be modified and which are not. 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 | The suite is designed to analyze and optimize software architectures, detect and prevent software defects and vulnerabilities, and provide insights into the code quality and maintainability. |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |
| Polyspace Bug Finder | R2023a | CERT C++: DCL52-CPP | Checks for:  const-qualified reference types  Modification of const-qualified reference types  Rule fully covered. |
| Helix QAC | 4.10 | C++0014 | Helix QAC provides a comprehensive set of coding standards, including MISRA C and C++, CERT C and C++, and others, and can be customized to enforce specific coding standards and requirements. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Data Value** | [STD-002-C]  INT30-C | A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type. This behavior is more informally called unsigned integer wrapping. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum = ui\_a + ui\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap: |
| #include <limits.h>    void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum;  if (UINT\_MAX - ui\_a < ui\_b) {  /\* Handle error \*/  } else {  usum = ui\_a + ui\_b;  }  /\* ... \*/  } |

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

| **Principles(s):** The principle that applies to INT30-C is the principle of integer overflow prevention. The INT30-C standard recommends that programmers should ensure that unsigned integer operations do not wrap. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Integer-overflow | Astrée is a static code analyzer that proves the absence of runtime errors and invalid concurrent behavior in safety-critical software written or generated in C or C++. |
| CodeSonar | 7.3p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | CodeSonar is a static code analysis tool from GrammaTech. CodeSonar is used to find and fix bugs and security vulnerabilities in source and binary code. It performs whole-program, inter-procedural analysis with abstract interpretation on C, C++, C#, Java, as well as x86 and ARM binary executables and libraries. |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Coverity® is a fast, accurate, and highly scalable static analysis (SAST) solution that helps development and security teams address security and quality defects early in the software development life cycle (SDLC), track and manage risks across the application portfolio, and ensure compliance with security and coding |
| LDRA tool suite | 9.7.1 | 493 S, 494 S | The LDRA tool suite is a comprehensive set of software standards compliance, testing, and verification tools based on industry best practices. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP]  STR50-CPP | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ std::basic\_string template requires no such character. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the unformatted input function std::basic\_istream<T>::read() is used to read an unformatted character array of 32 characters from the given file. However, the read() function does not guarantee that the string will be null terminated, so the subsequent call of the std::string constructor results in undefined behavior if the character array does not contain a null terminator. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  char buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }    std::string str(buffer);  // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the std::string object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use std::basic\_istream<T>::readsome() or a formatted input function, depending on need. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  char buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }  std::string str(buffer, in.gcount());  // ...  } |

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

| **Principles(s):** The principle behind this standard is to avoid conflicts between the names used in your code and those reserved by the C++ language or any libraries you are using. A standard from the "Secure Coding in C++" guidelines provided by CERT, which states that "Do not declare or define a reserved identifier or a reserved name." 10 |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube C/C++ Plugin | 4.10 | S3519 | SonarQube is an open platform to manage code quality. This plugin adds C++ support to SonarQube with the focus on integration of existing C++ tools. |
| RuleChecker | 22.10 | stream-input-char-array | RuleChecker primarily targets safety-critical embedded applications, but can also be used to analyze any structured programs, handwritten or generated, with complex memory usages, dynamic memory allocation, and recursion. |
| Polyspace Bug Finder | R2023a | CERT C++: STR50-CPP | Polyspace Bug Finder™ identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software. Using static analysis, including semantic analysis, Polyspace Bug Finder analyzes software control flow, data flow, and interprocedural behavior. |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Parasoft C/C++test checks code for compliance with a variety of functional safety, security, and coding standards. It also automatically generates the documentation needed to demonstrate compliance. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Define and use a pointer validation function** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C]  MEM10-C | Many functions accept pointers as arguments. If the function dereferences an invalid pointer (as in EXP34-C. Do not dereference null pointers) or reads or writes to a pointer that does not refer to an object, the results are undefined. Typically, the program will terminate abnormally when an invalid pointer is dereferenced, but it is possible for an invalid pointer to be dereferenced and its memory changed without abnormal termination [Jack 2007]. Such programs can be difficult to debug because of the difficulty in determining if a pointer is valid.  One way to eliminate invalid pointers is to define a function that accepts a pointer argument and indicates whether or not the pointer is valid for some definition of valid. For example, the following function declares any pointer to be valid except NULL: |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the incr() function increments the value referenced by its argument. It also ensures that its argument is not a null pointer. But the pointer could still be invalid, causing the function to corrupt memory or terminate abnormally. |
| void incr(int \*intptr) {  if (intptr == NULL) {  /\* Handle error \*/  }  (\*intptr)++;  } |

| **Compliant Code** |
| --- |
| This incr() function can be improved by using the valid() function. The resulting implementation is less likely to dereference an invalid pointer or write to memory that is outside the bounds of a valid object. |
| void incr(int \*intptr) {  if (!valid(intptr)) {  /\* Handle error \*/  }  (\*intptr)++;  } |

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

| **Principles(s):** 1, 10,8 principal maps to the idea of validating pointer arguments passed to functions to prevent undefined behavior. Preventative |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 159 S | The LDRA tool suite is a comprehensive set of software standards compliance, testing, and verification tools based on industry best practices. |
| Coverity | 2017.07 | CHECKED\_RETURN  NULL\_RETURNS  REVERSE\_INULL  FORWARD\_NULL | Coverity® is a fast, accurate, and highly scalable static analysis (SAST) solution that helps development and security teams address security and quality defects early in the software development life cycle (SDLC), track and manage risks across the application portfolio, and ensure compliance with security and coding |
| Cppcheck | 1.66 | nullPointer, nullPointerDefaultArg, nullPointerRedundantCheck | Cppcheck is a static code analysis tool for the C and C++ programming languages. It is a versatile tool that can check non-standard code. |
| Helix QAC | 2023.1 | DF2810, DF2811, DF2812, DF2813 | Static Code Analysis for C and C++ Helix QAC is a static code analyzer that automatically scans code for violations (based on C and C++ coding rules). It enables development teams to detect defects earlier in development — when they're easier and less costly to fix. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP]  MEM52-CPP | The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly. Furthermore, operator new[] can throw an error of type std::bad\_array\_new\_length, a subclass of std::bad\_alloc, if the size argument passed to new is negative or excessively large.  When using the nonthrowing form, it is imperative to check that the return value is not nullptr before accessing the resulting pointer. When using either form, be sure to comply with ERR50-CPP. Do not abruptly terminate the program. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // Handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

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

| **Principles(s):** 10,9 the behavior of the default memory allocation operator, the non-throwing form of the allocation function, and the default allocator object is standardized in the C++ Standard, and their behavior is well-defined and consistent across different implementations of the C++ language. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.24 | V522, V668 | To check if the code is handling memory allocation errors correctly, PVS-Studio performs a static analysis of the code to detect potential errors related to memory allocation. PVS-Studio analyzes the code for potential issues such as uninitialized memory usage, memory leaks, null pointer dereference, and buffer overflows, which can result from improper handling of memory allocation errors. |
| Polyspace Bug Finder | R2023a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Parasoft Insure++ | [Insert text.] | [Insert text.] | Runtime detection |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Incorporate diagnostic tests using assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-C]  MSC11-C | Incorporate diagnostic tests into your program using, for example, the assert() macro. Assertions are primarily intended for use during debugging and are generally turned off before code is deployed by defining the NDEBUG macro (typically as a flag passed to the compiler). Consequently, assertions should be used to protect against incorrect programmer assumptions and not for runtime error checking. In particular, assertions are generally unsuitable for server programs or embedded systems in deployment. A failed assertion can lead to a denial-of-service attack if triggered by a malicious user, such as size being derived, in some way, from client input. In such situations, a soft failure mode, such as writing to a log file and rejecting the request, is more appropriate. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. See also MEM11-C. Do not assume infinite heap space and void MEM32-C. Detect and handle memory allocation errors. |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char \*)malloc(len + 1);  assert(NULL != dup);    memcpy(dup, c\_str, len + 1);  return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char\*)malloc(len + 1);  /\* Detect and handle memory allocation error \*/  if (NULL == dup) {  return NULL;  }    memcpy(dup, c\_str, len + 1);  return dup;  } |

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

| **Principles(s): 8,9 Incorporating diagnostic tests into a program using the assert() macro can help detect issues and improve program reliability during development. Assertions are particularly useful for checking programmer assumptions and detecting logical errors in the program.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.FUNCS.ASSERTS | check if the code is handling errors correctly, including memory allocation errors, CodeSonar performs a variety of checks that can help detect issues related to runtime errors and potential vulnerabilities. |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2022.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP]  ERR51-CPP | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The default terminate handler called by std::terminate() calls std::abort(), which abnormally terminates the process. When std::abort() is called, or if the implementation does not unwind the stack prior to calling std::terminate(), destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for denial-of-service attacks. For more information on implicitly calling std::terminate(), see ERR50-CPP. Do not abruptly terminate the program.  All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.  As per ERR50-CPP-EX1, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the main() function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion. |

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

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** 10,9 The ERR51-CPP coding guideline advises developers to handle all exceptions in their code. This means that all functions that could potentially throw an exception should have a try-catch block that handles the exception or propagates it to the caller.  Handling exceptions properly ensures that the program's execution does not terminate unexpectedly and allows the program to continue running in a predictable manner. Failure to handle exceptions can lead to undefined behavior, data corruption, and other serious issues that can impact the program's reliability and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | To detect violations of the ERR51-CPP guideline, CodeSonar performs a variety of analyses on the source code, including control flow analysis and data flow analysis. By analyzing the code's control flow, CodeSonar can identify areas of the code where exceptions can be thrown and ensure that each of these areas is covered by a corresponding catch block that handles the exception. |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2023a | CERT C++: ERR51-CPP | Checks for unhandled exceptions |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | RuleChecker primarily targets safety-critical embedded applications, but can also be used to analyze any structured programs, handwritten or generated, with complex memory usages, dynamic memory allocation, and recursion. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Arguments to character-handling functions must be representable as an unsigned char** |
| --- | --- | --- |
| **Characters** | [STD-008-C]  STR37-C | The header <ctype.h> declares several functions useful for classifying and mapping characters. In all cases the argument is an int, the value of which shall be representable as an unsigned char or shall equal the value of the macro EOF. If the argument has any other value, the behavior is undefined. This rule is applicable only to code that runs on platforms where the char data type is defined to have the same range, representation, and behavior as signed char. |

| **Noncompliant Code** |
| --- |
| On implementations where plain char is signed, this code example is noncompliant because the parameter to isspace(), \*t, is defined as a const char \*, and this value might not be representable as an unsigned char: |
| #include <ctype.h>  #include <string.h>    size\_t count\_preceding\_whitespace(const char \*s) {  const char \*t = s;  size\_t length = strlen(s) + 1;  while (isspace(\*t) && (t - s < length)) {  ++t;  }  return t - s;  } |

| **Compliant Code** |
| --- |
| This compliant solution casts the character to unsigned char before passing it as an argument to the isspace() function: |
| #include <ctype.h>  #include <string.h>    size\_t count\_preceding\_whitespace(const char \*s) {  const char \*t = s;  size\_t length = strlen(s) + 1;  while (isspace((unsigned char)\*t) && (t - s < length)) {  ++t;  }  return t - s;  } |

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

| **Principles(s):** 10 To ensure the safe and predictable behavior of these functions, the STR37-C guideline recommends that arguments passed to character-handling functions should be cast to an unsigned char before calling the function. This ensures that the argument is within the valid range of unsigned char values and avoids undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE | [Insert text.] | [Insert text.] | Could detect violations of this rule by seeing if the argument to a character handling function (listed above) is not an unsigned char |
| Parasoft C/C++test | 2022.2 | CERT\_C-STR37-a | Do not pass incorrect values to ctype.h library functions |
| Polyspace Bug Finder | R2023a | CERT C: Rule STR37-C | Checks for invalid use of standard library integer routine (rule fully covered) |
| ECLAIR | 1.2 | CC2.STR37 | ECLAIR works on the desktop to find critical defects while software is being coded, in the context of the build environment and compiler. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not rely on side effects in operands to sizeof, \_Alignof, or \_Generic** |
| --- | --- | --- |
| **Expressions** | [STD-009-C]  EXP44-C | Some operators do not evaluate their operands beyond the type information the operands provide. When using one of these operators, do not pass an operand that would otherwise yield a side effect since the side effect will not be generated.  The sizeof operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. In most cases, the operand is not evaluated. A possible exception is when the type of the operand is a variable length array type (VLA); then the expression is evaluated. When part of the operand of the sizeof operator is a VLA type and when changing the value of the VLA's size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. (See unspecified behavior 22.)  The operand passed to\_Alignof is never evaluated, despite not being an expression. For instance, if the operand is a VLA type and the VLA's size expression contains a side effect, that side effect is never evaluated.  The operand used in the controlling expression of a \_Generic selection expression is never evaluated.  Providing an expression that appears to produce side effects may be misleading to programmers who are not aware that these expressions are not evaluated, and in the case of a VLA used in sizeof, have unspecified results. As a result, programmers may make invalid assumptions about program state, leading to errors and possible software vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the expression a++ is not evaluated:  Consequently, the value of a after b has been initialized is 14. |
| #include <stdio.h>    void func(void) {  int a = 14;  int b = sizeof(a++);  printf("%d, %d\n", a, b);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the variable a is incremented outside of the sizeof operation: |
| #include <stdio.h>    void func(void) {  int a = 14;  int b = sizeof(a);  ++a;  printf("%d, %d\n", a, b);  } |

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

| **Principles(s):** 10,1 When using certain operators, such as sizeof, \_Alignof, and \_Generic, one should not pass an operand that may have side effects since such side effects may not be generated. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wunevaluated-expression | Can diagnose some instance of this rule, but not all (such as the \_Alignof NCCE) |
| CodeSonar | 7.3p0 | LANG.STRUCT.SE.SIZEOF | Side effects in sizeof |
| Parasoft C/C++test | 2022.2 | CERT\_C-EXP44-a  CERT\_C-EXP44-b | Object designated by a volatile lvalue should not be accessed in the operand of the sizeof operator  The function call that causes the side effect shall not be the operand of the sizeof operator |
| Polyspace Bug Finder | R2023a | CERT C: Rule EXP44-C | Checks for situations when side effects of specified expressions are ignored (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Guarantee exception safety** |
| --- | --- | --- |
| **Exceptions** | [STD-010-CPP]  ERR56-CPP | Proper handling of errors and exceptional situations is essential for the continued correct operation of software. The preferred mechanism for reporting errors in a C++ program is exceptions rather than error codes. A number of core language facilities, including dynamic\_cast, operator new(), and typeid, report failures by throwing exceptions. In addition, the C++ standard library makes heavy use of exceptions to report several different kinds of failures. Few C++ programs manage to avoid using some of these facilities. Consequently, the vast majority of C++ programs must be prepared for exceptions to occur and must handle each appropriately. (See ERR51-CPP. Handle all exceptions.)  Because exceptions introduce code paths into a program, it is important to consider the effects of code taking such paths and to avoid any undesirable effects that might arise otherwise. Some such effects include failure to release an acquired resource, thereby introducing a leak, and failure to reestablish a class invariant after a partial update to an object or even a partial object update while maintaining all invariants. Code that avoids any such undesirable effects is said to be exception safe. Code that guarantees strong exception safety also guarantees basic exception safety.  Because all exceptions thrown in an application must be handled, in compliance with ERR50-CPP. Do not abruptly terminate the program, it is critical that thrown exceptions do not leave the program in an indeterminate state where invariants are violated. That is, the program must provide basic exception safety for all invariants and may choose to provide strong exception safety for some invariants. Whether exception handling is used to control the termination of the program or to recover from an exceptional situation, a violated invariant leaves the program in a state where graceful continued execution is likely to introduce security vulnerabilities. Thus, code that provides no exception safety guarantee is unsafe and must be considered defective. |

| **Noncompliant Code** |
| --- |
| The following noncompliant code example shows a flawed copy assignment operator. The implicit invariants of the class are that the array member is a valid (possibly null) pointer and that the nElems member stores the number of elements in the array pointed to by array. The function deallocates array and assigns the element counter, nElems, before allocating a new block of memory for the copy. As a result, if the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior. |
| #include <cstring>    class IntArray {  int \*array;  std::size\_t nElems;  public:  // ...    ~IntArray() {  delete[] array;  }      IntArray(const IntArray& that); // nontrivial copy constructor  IntArray& operator=(const IntArray &rhs) {  if (this != &rhs) {  delete[] array;  array = nullptr;  nElems = rhs.nElems;  if (nElems) {  array = new int[nElems];  std::memcpy(array, rhs.array, nElems \* sizeof(\*array));  }  }  return \*this;  }    // ...  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the copy assignment operator provides the strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. Only after the allocation succeeds does the function proceed to change the state of the object. In addition, by copying the array to the newly allocated storage before deallocating the existing array, the function avoids the test for self-assignment, which improves the performance of the code in the common case [Sutter 2004]. |
| #include <cstring>    class IntArray {  int \*array;  std::size\_t nElems;  public:  // ...    ~IntArray() {  delete[] array;  }    IntArray(const IntArray& that); // nontrivial copy constructor    IntArray& operator=(const IntArray &rhs) {  int \*tmp = nullptr;  if (rhs.nElems) {  tmp = new int[rhs.nElems];  std::memcpy(tmp, rhs.array, rhs.nElems \* sizeof(\*array));  }  delete[] array;  array = tmp;  nElems = rhs.nElems;  return \*this;  }    // ...  }; |

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

| **Principles(s):** 10,4,2 The importance of proper handling of errors and exceptional situations in software. It emphasizes that the preferred mechanism for reporting errors in C++ programs is exceptions rather than error codes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions  Do not leave 'catch' blocks empty |
| Polyspace Bug Finder | R2023a | CERT C++: ERR56-CPP | Checks for exceptions violating class invariant (rule fully covered). |
| PRQA QA-C++ | 4.4 | 4075, 4076 | QA·C++ identifies problems in C++ source code caused by language usage that’s dangerous, overly complex, non-portable, difficult to maintain, or simply diverges from coding standards—you get comprehensive coverage |
| PVS-Studio | 7.24 | V565, V1023, V5002 | PVS‑Studio is a tool to detect bugs and potential vulnerabilities in C, C++, C#, and Java source code on Windows, Linux, macOS. PVS‑Studio provides a plugin to import analysis results into SonarQube. The plugin allows to import warnings generated by the PVS‑Studio analyzer into the SonarQube server database. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Insert your written explanations here.]

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Low | High | 2 |
| STD-002-C | High | Likely | High | Medium | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-C | High | Unlikely | High | Low | 3 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-C | Low | Unlikely | High | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-C | Low | Unlikely | Low | Low | 3 |
| STD-009-C | Low | Unlikely | Low | Low | 3 |
| STD-010-CPP | High | Likely | High | Medium | 2 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest is encryption that is used to help protect data that is stored on a disk or backup media. Encryption at rest is designed to prevent the attacker from accessing the unencrypted data by ensuring the data is encrypted when on disk. The same key encrypts and decrypts data. |
| Encryption at flight | Encrption at flight means that you encrypt data when it’s being transmitted. Data that goes over internal network or the internet is vulnerable. Each packet sent in a session is encrypted independently. Transport encryption uses two symmetric 128-bit AES session keys, randomly generated at session initiation. Dual-key encryption and decryption allows packet-by-packet verification at the remote end, and non-conforming packets are discarded (Vince Lo Faso, 2015). |
| Encryption in use | Encryption in use is the process of encrypting data while it is being used. This ensures sensitive data is never left unsecured. All sensitive data is encrypted, data is encrypted throughout the entire data life, access to unencrypted data is controlled, Governance is provided through a centralized, simple platform, Anomalies are detected and responded to in real time. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication identifies the user to ensure the right user is being granted access. This is normally done with username and password. If the username and the password match what is stored the user is granted access. |
| Authorization | Authorization is the process that determines whether the user has authority to issue certain commands or certain resources. Users should only have access to the resources needed to fulfill their task. Access should be denied once that resource is no longer needed, or task is complete. This also ensures someone without the proper authorization can not modify systems or databases. |
| Accounting | Accounting is a tool utilized to account for user behavior. Accounting can be used to find what resources were accessed and what activities were done on the users account. Accounting is carried out by logging session statistics and usage information and is used for authorization control, billing, trend analysis, resource utilization, and capacity planning activities. |

**\***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 | 4/7/2023 | Security Policy | Donald Thibodeaux |  |
| [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 |