**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

# Nicholas Feero | CS-405: Secure Coding

## 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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 20](#_Toc52464066)

[Coding Standard 9 22](#_Toc52464067)

[Coding Standard 10 25](#_Toc52464068)

[Defense-in-Depth Illustration 27](#_Toc52464069)

[Project One 27](#_Toc52464070)

[1. Revise the C/C++ Standards 27](#_Toc52464071)

[2. Risk Assessment 27](#_Toc52464072)

[3. Automated Detection 27](#_Toc52464073)

[4. Automation 27](#_Toc52464074)

[5. Summary of Risk Assessments 28](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 28](#_Toc52464076)

[7. Map the Principles 30](#_Toc52464077)

[Audit Controls and Management 31](#_Toc52464078)

[Enforcement 31](#_Toc52464079)

[Exceptions Process 31](#_Toc52464080)

[Distribution 32](#_Toc52464081)

[Policy Change Control 32](#_Toc52464082)

[Policy Version History 32](#_Toc52464083)

[Appendix A Lookups 32](#_Toc52464084)

[Approved C/C++ Language Acronyms 32](#_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 | Validating input data ensures that all data acquired through an input system (users, devices, etc.) is appropriate and safe. Part of this process involves checking that inputs conform to the type and range requirements for acceptable inputs. This prevents attacks such as malicious data injections and buffer overflow attempts. |
| 1. Heed Compiler Warnings | Compiler warnings inform the developer of risky or unintended code behavior that can lead to vulnerabilities. As compiler warnings are presented, developers should address and treat them as errors; warnings should be treated seriously and resolved promptly. Ignoring, quieting, or suppressing warnings may allow execution errors, bugs, and security issues to remain unresolved. |
| 1. Architect and Design for Security Policies | Security should be integrated into the design and architecture of a system from the beginning of development, not added into it later. This principle addresses a secure structure to a system by considering issues like threats, mitigations, and compliance from a foundational necessity. Alongside this approach, this principle enforces security policies like data encryption and user authentication methods when constructing the system. |
| 1. Keep It Simple | This principle refers to keeping the design of a system as simple and small as possible. Complex designs increase the likelihood of introducing errors in their implementation, configuration, and use. With greater complexity, more effort is needed to reach an appropriate level of security assurance. Thus, keeping the design simple leads to an easier and more maintainable system design. |
| 1. Default Deny | Access to resources or functions should, initially, be denied by default and specifically allowed when its implementation is necessary. Systems should not make any assumptions that users, functions, or resources have access to it. Strict permissions are needed to minimize unauthorized access and increase overall system security. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege refers to users, systems, and processes being permitted the minimum level of access necessary to perform their expected functions. If elevated permissions are required, they should only be accessed for the least amount of time required to successfully perform the privileged task. Using this approach reduces the opportunities of malicious attacks through methods of executing unpredictable code with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | When data is sent to other systems like databases, APIs, or external applications, it should be sanitized to prevent potentially harmful content from being sent or intercepted. This principle prevents injection attacks and ensures the systems receiving the data can do so safely to maintain intersystem security. |
| 1. Practice Defense in Depth | Practicing Defense in Depth (DiD) involves using multiple layers of security, both redundant and unique, to protect a system. This principle ensures that as many areas of security are addressed as possible, and each area of security has support if one layer fails or is breached. This practice increases the assurance across consumers that their sensitive data is safe. |
| 1. Use Effective Quality Assurance Techniques | Using effective quality assurance techniques like code reviews, unit testing, and penetration testing are good methods to identify security flaws prior to a software’s release. Performing these techniques can lead to more secure systems by ensuring the systems operate as expected under expected and malicious conditions. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves examining the development language and platform a system operates in, and following the best practices, guidelines, and rules for the circumstance. Once these details have been established, it’s expected to become familiar with the specific vulnerabilities of the language and platform to avoid them. Using a secure coding standard such as CERT can outline common mistakes and vulnerabilities to promote consistency and secure practices among developers. |

### 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.

**NOTE:** Each coding standard references from [SEI CERT C++ Coding Standard](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682) or [SEI CERT C Coding Standard](https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard).

#### Coding Standard 1

| **Coding Standard** | **Label** | **Do not write syntactically ambiguous declarations (DCL53-CPP)** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Syntactically ambiguous declarations can lead to unexpected program execution. This can result in increased maintenance risk, different interpretations from varying compilers or tools, and unexpected runtime behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, it appears to be a call to the Widget constructor to instantiate an object named *w* of type *Widget*. However, since the line is interpreted by the compiler as a function declaration, not an object instantiation, this leads to unexpected behavior since the syntactically ambiguous declaration is interpreted differently than intended. |
| #include <iostream>    **struct** Widget {    Widget() { std::cout << "Constructed" << std::endl; }  };    **void** f() {    Widget w();  } |

| **Compliant Code** |
| --- |
| In this compliant example, objects of *w1* and *w2* avoid syntactically ambiguous declarations by being specifically object declarations. This avoids the misinterpretation of function declaration when instantiating objects. |
| #include <iostream>    **struct** Widget {    Widget() { std::cout << "Constructed" << std::endl; }  };    **void** f() {    Widget w1; // Elide the parentheses    Widget w2{}; // Use direct initialization  } |

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

| **Principles(s):**  2. Heed Compiler Warnings   * Ambiguous declarations create compiler warnings that suggest misinterpretation is possible with the code. Treating the ambiguous declaration compiler warnings seriously can prevent bugs and vulnerabilities.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by writing unambiguous code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium (2) | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **LANG.STRUCT.DECL.FNEST** | Nested Function Declaration |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: DCL53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl53cpp.html) | Checks for declarations that can be confused between:   * Function and object declaration * Unnamed object or function parameter declaration   Rule fully covered. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not read uninitialized memory (EXP53-CPP)** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Reading memory that has never been assigned a valid value results in undefined behavior like program crashes, unknown values being applied to unintended situations, and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an uninitialized local variable is evaluated as part of an output expression. This will result in undefined behavior due to the lack of initialization prior to displaying the variable’s value in the output. |
| #include <iostream>    **void** f() {  **int** i;     std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the local variable is initialized prior to displaying the variable’s value in the output. |
| #include <iostream>    **void** f() {  **int** i = 0;     std::cout << i;  } |

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

| **Principles(s):**  2. Heed Compiler Warnings   * Having uninitialized memory (uninitialized variables) can produce compiler warnings. Addressing these warnings can prevent unexpected behavior from occurring.   9. Use Effective Quality Assurance Techniques   * Implementing practices like unit testing or static analysis can catch uninitialized memory from being released or from going unnoticed.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by initializing memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium (2) | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **uninitialized-read** | Partially checked |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -Wuninitialized clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **LANG.STRUCT.RPL LANG.MEM.UVAR** | Return pointer to local Uninitialized variable |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: EXP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp53cpp.html) | Checks for:   * Non-initialized variable * Non-initialized pointer   Rule partially covered. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator (STR50-CPP)** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Applying string data to a buffer that is too small to handle that data will result in a buffer overflow. This can lead to a condition where arbitrary code can be executed maliciously to gain unintended permissions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the buffer doesn’t guarantee that a null terminator is appended to the buffer and that the std::string str(buffer) is safe. This will result in undefined behavior if the character array doesn’t 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** |
| --- |
| In this compliant example, instead of appending a null terminator, it constructs std::string based on the number of characters read from the input stream. Thus, an appropriate string can be read from the input stream with sufficient space for all character data. |
| #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):**  1. Validate Input Data   * Ensuring storage for strings has sufficient space prevents vulnerabilities like buffer overflows from occurring.   7. Sanitize Data Sent to Other Systems   * Properly implementing and binding strings prevents weaknesses like injections from occurring when sent to other systems.   9. Use Effective Quality Assurance Techniques   * Utilizing testing techniques can determine weaknesses from insufficient storage allocation.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by allocating appropriate space for strings to prevent storage/memory weaknesses and exploitations. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low (1) | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO** | No space for null terminator Buffer overrun Type overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-STR50-b** **CERT\_CPP-STR50-c** **CERT\_CPP-STR50-e** **CERT\_CPP-STR50-f** **CERT\_CPP-STR50-g** | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: STR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr50cpp.html) | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems (STR02-C)** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | When failing to sanitize data passed to complex subsystems, potential injection attacks, data integrity issues, and breaches of sensitive data can occur. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, unsanitized data is passed from the “USER” source, an untrusted source, as an argument to the *login* program. This can lead to authentication bypasses, injection, and privilege risks. |
| (**void**) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")),    0); |

| **Compliant Code** |
| --- |
| In this compliant example, the “—” argument occurs before the call to the source “USER.” This instructs the system to stop interpreting options in the argument list, so input from the “USER” source can’t be utilized by an attacker. |
| (**void**) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")), 0); |

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

| **Principles(s):**  1. Validate Input Data   * Ensuring all data is valid and safe before it reaches a subsystem prevents malicious inputs.   7. Sanitize Data Sent to Other Systems   * Directly addresses validating and cleaning data passed to subsystems.   8. Practice Defense in Depth   * Data sanitization is included as a layer of security. This prevents data from being exploited or manipulated during the data transfer process.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by preventing injection and malicious data manipulation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low (1) | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.1p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C: Rec. STR02-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str02c.html) | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory (MEM50-CPP)** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Reading memory after it has been deallocated can lead to abnormal program behavior (program termination) and Denial-of-Service (DoS) attacks. Writing memory that has been deallocated can lead to arbitrary code executions leaving the system vulnerable. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, *s* is deallocated with delete *s*. After delete, the pointer *s* becomes a dangling pointer. After calling s->f(), the dangling pointer is dereferenced which results in undefined behavior. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant example, the call s->f() is made before delete *s*. Thus, memory is still valid while accessing *s.* After delete *s*, the pointer *s* is **not** used again which avoids dereferencing a dangling pointer. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |

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

| **Principles(s):**  2. Heed Compiler Warnings   * Potential compiler warnings can arise from use-after-free scenarios. Resolving these issues can prevent unexpected behavior from accessing freed memory.   8. Practice Defense in Depth   * Avoiding use-after-free scenarios can reduce the number of potential malicious attack points in your code. This serves as an additional layer of security to support a code’s integrity.   9. Use Effective Quality Assurance Techniques   * Certain testing tools can catch memory issues early in a code’s development. This can help keep your code maintainable and secure.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by limiting the exploitation of dangling pointers attempting to access freed memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low (1) | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **ALLOC.UAF** | Use after free |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | v7.5.0 | **USE\_AFTER\_FREE** | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: MEM50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem50cpp.html) | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression (DCL03-C)** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Static assertions are great as a diagnostic tool for finding and solving software defects that lead to vulnerabilities; however, they need to be incorporated correctly to not create clutter and unnecessary assertions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the use of the runtime assertion needs to be placed in a function and executed. In this example, assert only checks the condition during runtime and isn’t clearly within the function it’s attempting to assert. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| In this compliant example, it utilizes an expression that allows for evaluation at compile time rather than runtime. The static\_assert ensures that the condition is verified during compilation and not at runtime. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    static\_assert(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**),                "Structure must not have any padding"); |

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

| **Principles(s):**  2. Heed Compiler Warnings   * Static assertions can catch invalid assumptions from a constant expression, which typically results in compiler warnings.   9. Use Effective Quality Assurance Techniques   * Static assertions are a proactive quality tool that allows developers to detect issues early and reduce vulnerabilities.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by checking if assertion use is appropriate, maintainable, and safe. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High (3) | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.1p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Do not leak resources when handling exceptions (ERR57-CPP)** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Resource leaks will eventually cause a program to crash. With attackers being able to continuously force exceptions to be thrown, repeated resource leaks can occur. This allows attackers to perform a Denial-of-Service (DoS) attack. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an object *pst* is not properly released when an exception is thrown. This causes a resource leak since the object’s data isn’t being deleted while an exception is thrown. |
| #include <new>    **struct** SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.  **void** process\_item() noexcept(**false**);  };    **void** f() {    SomeType \*pst = **new** (std::**nothrow**) SomeType();  **if** (!pst) {      // Handle error  **return**;    }    **try** {      pst->process\_item();    } **catch** (...) {      // Process error, but do not recover from it; rethrow.  **throw**;    }  **delete** pst;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the object *pst* is released by calling delete within the exception. This prevents a resource leak since the object’s data is being released while an exception block is being executed. |
| #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):**  4. Keep It Simple   * Streamlining and simplifying the use of resources, methods, and calls when handling exceptions reduces the complexity of the code and the potential for leaks.   8. Practice Defense in Depth   * When exceptions occur, ensuring the appropriate resources are released will provide a layer of security in preventing malicious attempts at unreleased resources.   9. Use Effective Quality Assurance Techniques   * Utilizing testing techniques and code reviews can help in determining unused or non-functional exceptions that can lead to potential leaks in resources.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by utilizing safe exception handling methods. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Low (1) | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **ALLOC.LEAK** | Leak |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-ERR57-a** | Ensure resources are freed |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: ERR57-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr57cpp.html) | Checks for:   * Resource leak caused by exception * Object left in partially initialized state * Bad allocation in constructor |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Write constructor member initializers in the canonical order (OOP53-CPP)** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Writing member initializers in a non-canonical order can result in undefined behavior which leads to reading uninitialized memory. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the member initializer attempts to initialize *someVal* first and *dependsOnSomeVal* to a value dependent on *someVal*. Since the declaration order of member variables doesn’t match the member initializer order, attempting to read the value of *someVal* results in an unspecified value being stored into *dependsOnSomeVal*. |
| **class** C {  **int** dependsOnSomeVal;  **int** someVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| In this compliant example, the declaration order of the member variables is changed so that the member initializer is ordered properly to avoid an unspecified value from being stored. |
| **class** C {  **int** someVal;  **int** dependsOnSomeVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):**  2. Heed Compiler Warnings   * When constructor member initializers are not in canonical order, compiler warnings will often occur. Treating these warnings seriously and resolving them will prevent unexpected behavior.   9. Use Effective Quality Assurance Techniques   * Utilizing tools like static analysis and performing code reviews can help catch non-canonical initializers.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by having appropriate and predictable initializers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | High (3) | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **initializer-list-order** | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **LANG.STRUCT.INIT.OOMI** | Out of Order Member Initializers |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-OOP53-a** | List members in an initialization list in the order in which they are declared |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: OOP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcoop53cpp.html) | Checks for members not initialized in canonical order (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Preserve thread safety and liveness when using condition variables (CON55-CPP)** |
| --- | --- | --- |
| Concurrency | [STD-009-CPP] | Failure to preserve thread safety and liveness when using condition variables can lead to repeatedly denied access of a needed resource and Denial-of-Service (DoS). |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the code is susceptible to a missed signal problem which occurs when the condition is checked before the thread goes to sleep. This sequence of actions results in a deadlock. |
| #include <condition\_variable>  #include <iostream>  #include <mutex>  #include <thread>    std::mutex mutex;  std::condition\_variable cond;    **void** run\_step(**size\_t** myStep) {  **static** **size\_t** currentStep = 0;    std::unique\_lock<std::mutex> lk(mutex);      std::cout << "Thread " << myStep << " has the lock" << std::endl;    **while** (currentStep != myStep) {      std::cout << "Thread " << myStep << " is sleeping..." << std::endl;      cond.wait(lk);      std::cout << "Thread " << myStep << " woke up" << std::endl;    }      // Do processing...    std::cout << "Thread " << myStep << " is processing..." << std::endl;    currentStep++;      // Signal awaiting task.    cond.notify\_one();      std::cout << "Thread " << myStep << " is exiting..." << std::endl;  }    **int** main() {    constexpr **size\_t** numThreads = 5;    std::**thread** threads[numThreads];      // Create threads.  **for** (**size\_t** i = 0; i < numThreads; ++i) {      threads[i] = std::**thread**(run\_step, i);    }      // Wait for all threads to complete.  **for** (**size\_t** i = numThreads; i != 0; --i) {      threads[i - 1].join();    }  } |

| **Compliant Code** |
| --- |
| In this compliant example, the code avoids missed signals by having dedicated condition variables in each thread. This ensures that each thread only waits for its own unique condition variable. |
| #include <condition\_variable>  #include <iostream>  #include <mutex>  #include <thread>    constexpr **size\_t** numThreads = 5;    std::mutex mutex;  std::condition\_variable cond[numThreads];    **void** run\_step(**size\_t** myStep) {  **static** **size\_t** currentStep = 0;    std::unique\_lock<std::mutex> lk(mutex);      std::cout << "Thread " << myStep << " has the lock" << std::endl;    **while** (currentStep != myStep) {      std::cout << "Thread " << myStep << " is sleeping..." << std::endl;      cond[myStep].wait(lk);      std::cout << "Thread " << myStep << " woke up" << std::endl;    }      // Do processing ...    std::cout << "Thread " << myStep << " is processing..." << std::endl;    currentStep++;      // Signal next step thread.  **if** ((myStep + 1) < numThreads) {      cond[myStep + 1].notify\_one();    }      std::cout << "Thread " << myStep << " is exiting..." << std::endl;  }    // ... main() unchanged ... |

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

| **Principles(s):**  4. Keep It Simple   * Being simple, clear, and concise when using condition variables reduces vulnerabilities and deadlocks.   8. Practice Defense in Depth   * Implementing proper thread safety adds an additional layer of security against concurrency issues.   9. Use Effective Quality Assurance Techniques   * Utilizing testing techniques like stress testing can catch liveness problems with concurrency issues.   10. Adopt a Secure Coding Standard   * Following CERT guidelines, this ensures expected behavior occurs by avoiding weakness in concurrency issues that can lead to liveness and thread safety threats. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium (2) | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **CONCURRENCY.BADFUNC.CNDSIGNAL** | Use of Condition Variable Signal |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-CON55-a** | Do not use the 'notify\_one()' function when multiple threads are waiting on the same condition variable |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: CON55-CPP](https://www.mathworks.com/help/bugfinder/ref/certccon55cpp.html) | Checks for multiple threads waiting for same condition variable (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Close files when they are no longer needed (FIO51-CPP)** |
| --- | --- | --- |
| Input Output (I/O) | [STD-010-CPP] | Failure to close files when they are no longer needed may allow attacks to exhaust the resources of a system, which increases the risk that written data won’t be cleared from in-memory file buffers if unexpected behavior occurs. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an abrupt termination call prevents an appropriate closure of file when no longer needed. This method bypasses normal object destruction methods for the std::fstream file. Thus, the close file method is never called and can result in the file remaining in an undefined state. |
| #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 compliant example, an object destruction method, file.close(), is called before std::terminate() to ensure the resources are closed and released prior to termination. Due to this structure, the use of std::terminate() is no longer abrupt and is safe to use since the file is already closed when this is called. |
| #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):**  4. Keep It Simple   * Managing files appropriately by closing them when they are no longer needed creates a simpler system design and reduces the likelihood of errors.   8. Practice Defense in Depth   * Ensuring appropriate resource release reduces potential threat routes and reduces resource exhaustion risks.   9. Use Effective Quality Assurance Techniques   * Utilizing techniques like code reviews and testing can help catch weaknesses in file handling.   10. Adopt a Secure Coding Standard   * Following secure coding guidelines for resource management, this ensures expected behavior occurs by preventing data integrity issues. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low (1) | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **ALLOC.LEAK** | Leak |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: FIO51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio51cpp.html) | Checks for resource leak (rule partially covered) |
| [Security Reviewer - Static Reviewer](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Security+Reviewer+-+Static+Reviewer) | 6.02 | **C80** | Fully implemented |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

[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 | Medium (2) | P2 | L3 |
| STD-002-CPP | High | Probable | Medium (2) | P12 | L1 |
| STD-003-CPP | High | Likely | Low (1) | P9 | L2 |
| STD-004-CPP | High | Likely | Low (1) | P9 | L2 |
| STD-005-CPP | High | Likely | Low (1) | P9 | L2 |
| STD-006-CPP | Low | Unlikely | High (3) | P3 | L3 |
| STD-007-CPP | Low | Probable | Low (1) | P2 | L3 |
| STD-008-CPP | Medium | Unlikely | High (3) | P6 | L2 |
| STD-009-CPP | Low | Unlikely | Medium (2) | P2 | L3 |
| STD-010-CPP | Medium | Unlikely | Low (1) | P2 | 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 at rest | Policy: Sensitive data that is being stored in any capacity (servers, databases, systems, etc.) must be encrypted by utilizing a proven encryption algorithm with secure key management.  Explanation: Encryption at rest protects stored data by converting it into an unreadable format that requires a specific key to decrypt. This should be applied in practice by implementing encryption across the entirety of a storage method and storing the encryption keys in a dedicated, secure location. Encryption at rest should be used to prevent data breaches if a storage method is compromised. This prevents data from being stolen and distributed. |
| Encryption in flight | Policy: Sensitive data being transmitted must be encrypted using a secure method to protect against interception.  Explanation: Encryption in flight protects data as it moves between systems, networks, applications, or users by converting it into an unreadable format while in transmission. This should be applied in practice by configuring the necessary services to handle encrypted transmission, securing connections, and using encrypted channels with APIs and remote communication. Encryption in flight should be used to prevent attacks from intercepting or manipulating data during transmission. |
| Encryption in use | Policy: Sensitive data actively being processed in memory must be protected with encryption techniques and encryption-handling environments.  Explanation: Encryption in use protects data while it is being processed in memory. This should be applied in practice by encrypting data fields so sensitive data can continue being encrypted while in memory. Encryption in use should be used to reduce the risk of data breaches that involve attacks on system memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Policy: Users, devices, and software must prompt a secure verification of identity before granting access to a system or data.  Explanation: Authentication is the process of confirming a user’s identity before granting access to data or a system. This should be applied in practice by enforcing the use of authentication methods like multi-factor authentication (MFA) to access sensitive data and systems, implementing secure password requirements, and managing logs to track sensitive information access. Authentication should be used to verify only expected and verified users are accessing systems with sensitive information. This reduces the risk of unauthenticated access which can result in stolen and distributed data. |
| Authorization | Policy: Users, systems, and processes are only granted the minimum necessary privileges to perform the expected actions of their role.  Explanation: Authorization is the process of determining what an authenticated source (user, system, etc.) is allowed to do and strictly limiting the source from performing any actions outside of its given permissions. This should be applied in practice by implementing role-based access to sensitive data and systems, performing frequent permission reviews, and examining data logs to ensure specific roles are accessing the appropriate data. Authorization should be used to prevent privilege escalation attacks and mitigate harm if a user’s account is compromised. |
| Accounting | Policy: All information regarding a system’s activity must be logged and retained for a defined period and reviewed frequently.  Explanation: Accounting is the process of recording user and system activity for examination. This should be applied in practice by tracking user actions, system changes, and data access while monitoring logs for unusual behavior. Accounting should be used to detect unusual behavior due to a malicious attack, or to track suspicious behavior and policy violations. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 07/20/2025 | First Draft (Module #3 Milestone) | Nicholas Feero |  |
| 1.2 | 08/10/2025 | Final Draft (Module #6 Project One) | Nicholas Feero |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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