**Green Pace Developer: Security Policy Guide Template Brennon Fultz CS 405**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Always validate input from untrusted sources, which should include most external sources. Adhering to this principle will prevent most vulnerabilities (Seacord, 2005) For example, use parameterized queries for interacting with a database. |
| 1. Heed Compiler Warnings | Detect and fix security flaws by adhering to the highest level of compiler warnings. Change the code to eliminate the warnings and consequently the vulnerability. For example, update deprecated libraries rather than suppressing warnings. |
| 1. Architect and Design for Security Policies | Build security into architecture decisions, ensuring the system allows for security mechanisms to be implemented. For example, take a separation of concerns approach to design to enable security checks between entities. |
| 1. Keep It Simple | Complex designs introduce potential points of failure and vulnerability and increase the effort required to assure security (Seacord, 2005). Use simple designs when possible. For example, keep session data stored locally and temporarily rather than in a database. |
| 1. Default Deny | The default option should always be to prevent access. Granting permission should be the basis of the security approach. For example, user accounts should by default have read-only permission. |
| 1. Adhere to the Principle of Least Privilege | Only grant permission to entities who absolutely must have it and only for so long as they must have it. This applies to software entities as well as users. For example, an API should only have routes defined for the actions it needs. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data sent to external systems to prevent unauthorized interactions (Seacord, 2005). For example, command line arguments passed out to automate a process should be sanitized. |
| 1. Practice Defense in Depth | Apply layers of security which overlap to prevent any single failure from allowing full access. Different types of defense will require different tactics to breach, increasing the difficulty of unauthorized access. For example, multi-factor authentication and role-based authorization. |
| 1. Use Effective Quality Assurance Techniques | Prevent vulnerabilities by producing high quality software. Use a variety of testing and QA techniques to reduce failures. For example, 3rd party penetration testing on a secured system. |
| 1. Adopt a Secure Coding Standard | Create or adhere to a security standard unique to your language or stack. For example, the SEI CERT C++ coding standard. |

### 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** | **Do not use floating point variables as loop counters** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | The precision limitations of floating point variables can lead to unexpected behavior, particularly when ported to various platforms (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses a floating point loop counter which will iterate an unclear number of times. |
| **void** func(**void**) {  **for** (**float** x = 0.1f; x <= 1.0f; x += 0.1f) {      /\* Loop may iterate 9 or 10 times \*/    }  } |

| **Compliant Code** |
| --- |
| This code uses the counter integer value to derive a floating point value within the loop while maintaining predictable behavior. |
| **void** func(**void**) {  **for** (**size\_t** count = 1; count <= 10; ++count) {  **float** x = count / 10.0f;      /\* Loop iterates exactly 10 times \*/    }  } |

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

| **Principles(s):** 4.) Keep it Simple: Use simple data types like int or size\_t as loop counters rather than floating-point types.  9.) Use Effective QA Techniques: A floating point loop counter may cause unintended behavior that should be fixed in QA. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | **for-loop-float** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-FLP30** | Fully implemented |
| Clang | 3.9 | cert-flp30-c | Checked by clang-tidy |
| CodeSonar | 8.3p0 | **LANG.STRUCT.LOOP.FPC** | Float-typed loop counter |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not pass a non-null terminated character sequence to a library function that expects a string** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Passing a non-null terminated character sequence to a library function that expects null termination can allow access to memory outside the bounds of the object (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code initializes c\_str[3] which is not null-terminated and will be passed to printf(). |
| #include <stdio.h>    **void** func(**void**) {  **char** c\_str[3] = "abc";  **printf**("%s\n", c\_str);  } |

| **Compliant Code** |
| --- |
| By removing the size in the declaration, the compiler will store the null-terminating character along with the string literal. |
| #include <stdio.h>    **void** func(**void**) {  **char** c\_str[] = "abc";  **printf**("%s\n", c\_str);  } |

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

| **Principles(s):** 7.) Sanitize Data Sent to Other Systems: Ensure library functions are used securely by passing the correct parameters. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-STR32** | Partially implemented: can detect some violation of the rule |
| CodeSonar | 8.3p0 | **MISC.MEM.NTERM.CSTRING** | Unterminated C String |
| LDRA tool suite | 9.7.1 | **404 S, 600 S** | Partially implemented |
| Parasoft C/C++test | 2024.2 | **CERT\_C-STR32-a** | Avoid overflow due to reading a not zero terminated string |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings can hold the null terminator** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Copying data to a buffer that is too small results in buffer overflow. Ensure the data destination has enough space for all characters and the null terminator (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| The lack of input bounds could cause an overflow in this noncompliant code. |
| **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Using std::string prevents buffer overflow by ensuring there is enough memory to hold the data in this compliant code. |
| **void** f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** 1.) Validate Input Data: designate adequate storage for input to prevent buffer overflow-based attacks. 9.) Use Effective Quality Assurance Techniques: Test boundary conditions to ensure overflow vulnerabilities are caught. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| CodeSonar | 8.3p0 | **MISC.MEM.NTERM** **LANG.MEM.BO** **LANG.MEM.TO** | No space for null terminator Buffer overrun Type overrun |
| RuleChecker | 22.10 | **stream-input-char-array** | Partially checked |
| Helix QAC | 2024.4 | **C++5216, DF2835, DF2836, DF2839,** | No space for null terminator |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Using user input in format strings can allow SQl injection among other attacks. This could lead to a crash or arbitrary code execution (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code directly uses the username in the format string output, which comes from an untrusted source. This is vulnerable to SQL injection. |
| **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fputs**(msg, stderr);  **free**(msg);  } |

| **Compliant Code** |
| --- |
| This compliant code passes the untrusted data to fprintf() rather than as part of the format string. |
| **void** incorrect\_password(**const** **char** \*user) {  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **fprintf**(stderr, msg\_format, user);  } |

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

| **Principles(s):** 1.) Validate Input Data: Data from untrusted sources should be validated to prevent SQL injection.  7.) Sanitize Data Sent to Other Systems: Data passed as output should be sanitized to prevent unauthorized interactions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-FIO30** | Partially implemented |
| CodeSonar | 8.3p0 | **IO.INJ.FMT** **MISC.FMT** | Format string injection Format string |
| Coverity | 2017.07 | **TAINTED\_STRING** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Pointers to deallocated memory (dangling pointers) can cause undefined behavior and create vulnerabilities (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code attempts to access s after it has been deleted, creating a vulnerability. |
| **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| This solution only deletes s after it is no longer needed, preventing access to deallocated memory. |
| **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):** 9.) Use Effective Quality Assurance Techniques: Dangling pointers should be fixed in QA so that they are not accessed erroneously.  3.) Architect and Design for Security Policies: Design the code such that memory is freed only after it is no longer needed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | **483 S, 484 S** | Partially implemented |
| Parasoft C/C++test | 2024.2 | **CERT\_CPP-MEM50-a** | Do not use resources that have been freed |
| Parasoft Insure++ |  |  | Runtime detection |
| CodeSonar | 8.3p0 | **ALLOC.UAF** | Use after free |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use assertions only to incorporate diagnostic tests** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions should be used to protect against incorrect assumptions about program behavior (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code is used during runtime to check if memory allocation was successful, which could lead to process termination. |
| **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 code does not use assert() at runtime to detect memory depletion. |
| **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):** 9.) Use Effective Quality Assurance Techniques: Assert statements should be limited to QA/Testing purposes.  3.) Architect and Design for Security Policies: Using assert() in non-test methods is a design anti-pattern that could cause unintended behavior.. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Tool | Version | Checker | Description |
| CodeSonar | 8.3p0 | **LANG.FUNCS.ASSERTS** | Not enough assertions |
| Coverity | 2017.07 | **ASSERT\_SIDE\_EFFECT** | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Unhandled exceptions can lead to abnormal process termination and denial-of-service vulnerability (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code throws an exception but does not catch any. This could lead to DoS vulnerability. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| This solution catches and handles the thrown exception. |
| **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):** 3.) Architect and Design for Security Policies: The code architecture should include robust exception handling and specific error messages, as well as secure failure mechanisms. Generic catch-all exception handling, or error-hiding, is a security anti-pattern that should be avoided. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | **527 S** | Partially implemented |
| Parasoft C/C++test | 2024.2 | **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 | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | **main-function-catch-all** **early-catch-all** | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Ensure unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Data Value** | [STD-008-CPP] | Unsigned integer operations that lead to wrapping can cause unexpected behavior and vulnerabilities (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code is subject to potential unsigned integer wrapping if the resulting usum is too large. |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a conditional to check if the resulting usum would be too large, thus preventing unexpected wrapping. |
| **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):** 3.) Architect and Design for Security Policies: Designs should include verification that unsigned integers do not wrap, ensuring that data value security is included from the start. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | **integer-overflow** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-INT30** | Implemented |
| CodeSonar | 8.3p0 | **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** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Coverity | 2017.07 | **INTEGER\_OVERFLOW** | Implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not attempt to modify a string literal** |
| --- | --- | --- |
| **String Correctness** | [STD-009-CPP] | Modifying a string literal is undefined behavior and may cause vulnerabilities related to access violations (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code initializes a string literal and then attempts to modify it, causing undefined behavior. |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| This compliant code uses an array initializer to store the values, so the modification is performed on a copy of the string literal, preventing undefined behavior. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** 9.) Use Effective Quality Assurance Techniques: Modifying a string literal causes undefined behavior and should be avoided as part of comprehensive QA.  3.) Architect and Design for Security Policies: Do not include code in the design that relies on modifying string literals, as this is a security anti-pattern. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | **string-literal-modfication write-to-string-literal** | Fully checked |
| Parasoft C/C++test | 2024.2 | **CERT\_C-STR30-a CERT\_C-STR30-b** | A string literal shall not be modified Do not modify string literals |
| PC-lint Plus | 1.4 | **489, 1776** | Partially supported |
| Polyspace Bug Finder | R2024a | CERT C: Rule STR30-C | Checks for writing to const qualified object (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-010-CPP] | All reference types are treated as const, therefore any const qualifier is ignored. CV-qualifying a reference type can cause undefined behavior (Seacord, 2005). |

| **Noncompliant Code** |
| --- |
| This noncompliant code declares p as const, then attempts to modify it. |
| **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 example removes the const qualifier from the reference type p. |
| **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):** 4.) Keep it Simple: Qualifying a reference type with const or volatile is unnecessary and may cause unintended behavior. Take a simpler approach by not qualifying reference types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |
| Polyspace Bug Finder | R2024a | CERT C++: DCL52-CPP | Checks for:  const-qualified reference types  Modification of const-qualified reference types  Rule fully covered. |
| Clang | 3.9 |  | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options. |

### 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.

Automation of the enforcement of these coding standards will become part of the development lifecycle as the DevOps process is converted to DevSecOps. Adding security to the automation pipeline will support security at each stage of development in accordance with the defense-in-depth principle. In the design and build stage, CI/CD automation tools such as Jenkins can implement dependency checking like OWASP to verify the security of third-party libraries as required by the “Architect and Design for Security” principle. The main enforcement of this policy will be automated in the build stage before testing, with Astree and any other necessary security checkers added to the pipeline to ensure compliance. Other static analysis tools such as RuleChecker will be used in the Verify and Test stage to ensure compliance and identify other vulnerabilities not defined in this policy. In production, LDAP and AP will be used to manage roles and role-based access control automatically as well as logs in the Monitor and Detect stage.

### 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 | Probable | Low | **P6** | **L2** |
| STD-002-CPP | High | Probable | Medium | **P12** | **L1** |
| STD-003-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-004-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-005-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-006-CPP | Low | Unlikely | High | **P1** | **L3** |
| STD-007-CPP | Low | Probable | Medium | **P4** | **L3** |
| STD-008-CPP | High | Likely | High | **P9** | **L2** |
| STD-009-CPP | Low | Likely | Low | **P9** | **L2** |
| STD-010-CPP | Low | Unlikely | Low | **P3** | **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 | Data at rest refers to data that is in storage, for example in a database or the cloud. Data at rest must be kept secure, particularly because the amount of data is much larger than data in flight or in use. Large data breaches that affect millions of users occur on data at rest. Encrypting such data is performed by passing the data through an encryption algorithm, such as SHA-256, before it is stored, so that only the encrypted form is kept at rest. This should always be done so that a breach will not allow attackers to access usable data. |
| Encryption in flight | Data in flight is any data that is being transmitted from one environment to another over a network, such as when data is accessed across the internet. Data in flight is particularly vulnerable because it is susceptible to various types of man-in-the-middle attacks which allow attackers to intercept data. For this reason, data in flight should be encrypted before being transmitted over a network and decrypted on arrival at the recipient. This should always be practiced, even on internal networks because those too can be accessed by a malicious third party. |
| Encryption in use | Data in use is exactly how it sounds – data that is being actively used. As such, it is necessarily vulnerable because it must be unencrypted to be of any use to the user or program that needs to access it, which makes this data susceptible to threats. However, this challenge can be mitigated in some cases by encrypting and decrypting data at runtime, so that it is only unencrypted for so lang as it needs to be. There is also the practice of homomorphic encryption which can enable operations to be performed on data that is still encrypted. Highly sensitive data should be accessed with encryption in-use to prevent breaches. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the practice of proving one’s identity before accessing a secure system to prevent improper access. For example, entering a username and password into a website tells the site that the user is the authentic owner of that account. Two-factor authentication should be used to log in to any company account. |
| Authorization | Authorization is the delegation of permissions based on who the user is. For example, an IT admin may have authorization to modify or add user accounts while normal users do not. This is known as role-based access control or RBAC and should be implemented with company accounts. Users should only have authorization to perform the essential functions of their role and modification of such authorization should be vetted and approved. |
| Accounting | Accounting is the practice of maintaining accurate records of what actions are performed, when, and by whom. For example, a database access record would show any users who viewed or modified data and when. This enables the organization to track data or network access and investigate suspicious activity or respond to critical events such as a breach. |

**\***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 | 01/25/2025 | Coding standards milestone. | Brennon Fultz |  |
| 2.0 | 02/09/2025 | Project One | Brennon Fultz |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

References

Seacord, R. 2005. *Top 10 Secure Coding Principles.* CMU Software Engineering Institute. Web. https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCom mentId=88044413

Seacord, R. 2005. *SEI CERT C++ Coding Standard.* CMU Software Engineering Institute. Web. https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCom mentId=88044413