**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 | Input data should be examined and tested to satisfy constraints before being allowed to interact with the database. All data should be processed to prevent malformed or malicious data from entering a system. |
| 1. Heed Compiler Warnings | While a compiler warning will not prevent code from compiling like an error will, these warning still exist to warn developers of potential errors or vulnerabilities within the code. These warnings should not be ignored and are there to help developers identify potential vulnerabilities that are more difficult to identify at runtime. |
| 1. Architect and Design for Security Policies | Before implementing any security policies, the overall architecture and design of the software system should be carefully studied and considered. This may include practices such as using the principle of least privilege, properly outlining authorization levels, and properly dividing the system into sub-systems. This information will be included in the security policy statement. |
| 1. Keep It Simple | System design should be as simple as possible to reduce the likelihood of errors and vulnerabilities. This makes development, maintenance, and security less difficult to manage. |
| 1. Default Deny | Access should be denied by default, and access should only be permitted once specific criteria has been met. This reduces the chances of malicious activity and limits the number of actors within any given component. |
| 1. Adhere to the Principle of Least Privilege | A user should be granted the minimum required privileges required to complete the desired task. Elevated access should only be granted if required for a specific task, and only for the time required to complete the task. This prevents users from being able to act outside of their scope. |
| 1. Sanitize Data Sent to Other Systems | Once taken in, data should be cleaned, which involves stripping unneeded or potentially harmful characters or spaces. This prevents actions like SQL injection, and all data should be cleaned prior before it is passed into another component. |
| 1. Practice Defense in Depth | Defense in depth refers to a system containing multiple layers of security, which reduces the likelihood of successful attacks because each layer has different strengths. If a single layer fails, there are other layers still in place. |
| 1. Use Effective Quality Assurance Techniques | QA testing should be completed early and often, and system requirements should be communicated clearly and precisely. Testing should be done at the unit, integration, and system level and should encompass non-functional, performance, and security parameters. Specific examples may include unit testing, code reviews, pen testing, and fuzz testing. |
| 1. Adopt a Secure Coding Standard | Developers should apply the specific coding standards in the given language and platform of choice to be secure from the start of development. |

### 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** | **Obey the one-definition rule** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly-defined. An inline function shall be defined in every translation unit in which it is odr-used. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two different translation units define a class of the same name with differing definitions. Although the two definitions are functionally equivalent (they both define a class named S with a single, public, nonstatic data member int a), they are not defined using the same sequence of tokens. |
| // a.cpp  **struct** S {  **int** a;  };    // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| The correct mitigation depends on programmer intent. If the programmer intends for the same class definition to be visible in both translation units because of common usage, the solution is to use a header file to introduce the object into both translation units, as shown in this compliant solution. |
| // S.h  **struct** S {  **int** a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** 2 – Heed Compiler Warnings, 4 – Keep it Simple, 10 – Adopt a Secure Coding Standard  Following the one-definition rule can map to heeding compiler warnings because the compiler will warn developers if classes or variables do not possess unique names. Additionally, keeping the program simple is related to this because variables and classes should be named in a descriptive fashion, and each element should only be named once. With simple code, there is less of a chance that elements will be incorrectly named or declared multiple times. Finally, adopting a secure coding standard will ensure that all developers on the team are following the same principles and they will each be aware of naming conventions to prevent duplicate definitions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **type-compatibility** **definition-duplicate** **undefined-extern** **undefined-extern-pure-virtual** **external-file-spreading** **type-file-spreading** | Partially checked |
| LDRA tool suite | 9.7.1 | **286 S, 287 S** | Fully implemented |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-DCL60-a** | A class, union or enum name (including qualification, if any) shall be a unique identifier |
| Polyspace Bug Finder | R2024a | CERT C++: DCL60-CPP | Checks for inline constraints not respected (rule partially covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not cast an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | The range of values that can be represented by enumerations may include enumerator values not specified by the enumeration itself. To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** 1 – Validate Input Data, 7 – Sanitize Data Sent to Other Systems, 9 – Use Effective Quality Assurance Techniques  Casting out-of-range enumeration values can result in buffer overflows and introduce major security concerns to a system. For data structures that are populated dynamically, all user input should be validated to ensure it is of the proper type and within the bounds for that given structure. Furthermore, all data should be validated before being sent to other subsystems or the database. Using assertions and effective quality assurance techniques will test for out-of-range values, and these errors will be caught in development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **cast-integer-to-enum** | Partially checked |
| CodeSonar | 8.1p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |
| RuleChecker | 22.10 | **cast-integer-to-enum** | Partially checked |
| Polyspace Bug Finder | R2024a | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior. |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** 4 – Keep it Simple, 9 - Use Effective Quality Assurance Techniques, 10 – Adopt a Secure Coding Standard  Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory, and this can potentially lead to unmanaged memory within stacks and heaps or dangling pointers. Keeping the program simple would result in string literals being created once and not altered throughout the program. Additionally, effective quality assurance testing would find this issue and adopting a secure coding standard would prevent this issue across a development team. Not catching these issues could potentially lead to widespread issues as a result of poorly managed memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | **157 S** | Partially implemented |
| Parasoft C/C++test | 2023.1 | **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 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted. |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data. |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):** 1 - Validate Input Data, 3 – Architect and Design for Security Policies, 5 – Default Deny, 6 – Adhere to the Principle of Least Privilege, 7 – Sanitize Data Sent to Other Systems  Based on the name of this standard, properly sanitizing data before sending it to complex subsystems directly relates to principles 1 and 7 because all data should be validated and sanitized before it is allowed to interact with other portions of the system. In addition, the system should be designed with security in mind, and developers should carefully consider how data will flow throughout the program and how all data should be validated to prevent vulnerabilities. Finally, if a user is attempting to send this data, default deny and the principle of least privilege will reduce the likelihood that malicious data is introduced to the system because standard users will not be allowed to send data throughout the system without the proper roles being assigned to them. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | **TAINTED\_STRING** | Fully implemented |
| LDRA tool suite | 9.7.1 | **108 D, 109 D** | Partially implemented |
| Parasoft C/C++test | 2023.1 | **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 | R2024a | CERT C: Rec. STR02-C | 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** | **Store a new value in pointers immediately after free()** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Dangling pointers can lead to exploitable double-free and access-freed-memory vulnerabilities. A simple yet effective way to eliminate dangling pointers and avoid many memory-related vulnerabilities is to set pointers to NULL after they are freed or to set them to another valid object. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the type of a message is used to determine how to process the message itself. It is assumed that message\_type is an integer and message is a pointer to an array of characters that were allocated dynamically. If message\_type equals value\_1, the message is processed accordingly. A similar operation occurs when message\_type equals value\_2. However, if message\_type == value\_1 evaluates to true and message\_type == value\_2 also evaluates to true, then message is freed twice, resulting in a double-free vulnerability. |
| **char** \*message;  **int** message\_type;    /\* Initialize message and message\_type \*/    **if** (message\_type == value\_1) {    /\* Process message type 1 \*/  **free**(message);  }  /\* ...\*/  **if** (message\_type == value\_2) {     /\* Process message type 2 \*/  **free**(message);  } |

| **Compliant Code** |
| --- |
| Calling free() on a null pointer results in no action being taken by free(). Setting message to NULL after it is freed eliminates the possibility that the message pointer can be used to free the same memory more than once. |
| **char** \*message;  **int** message\_type;    /\* Initialize message and message\_type \*/    **if** (message\_type == value\_1) {    /\* Process message type 1 \*/  **free**(message);    message = NULL;  }  /\* ... \*/  **if** (message\_type == value\_2) {    /\* Process message type 2 \*/  **free**(message);    message = NULL;  } |

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

| **Principles(s):** 3 - Architect and Design for Security Policies  If a system is being designed with security in mind, no dangling pointers should be left, because this could lead to security vulnerabilities. All pointers should be handled and freed correctly once not in use, which greatly reduces the risk of memory-related vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-MEM01** | Fully implemented |
| CodeSonar | 8.1p0 | **ALLOC.DF** **ALLOC.UAF** | Double free Use after free |
| Coverity | 2017.07 | **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 |
| LDRA tool suite | 9.7.1 | **484 S, 112 D** | Partially implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, or a portable compliant solution uses static\_assert: |
| #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):** 8 – Practice Defense in Depth, 9 – Use Effective Quality Assurance Techniques, 10 - Adopt a Secure Coding Standard  The proper use of assertions is a critical aspect of testing software and ensuring security. Assertions can be layered throughout a system to follow the principle of defense in depth. Furthermore, these tools can be used as a part of effective quality assurance. Additionally, it should be the standard for a development team should design and implement assertions in a development environment. This will help prevent sever vulnerabilities as the system grows and components integrate. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.1p0 | **customization** | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | **CC2.DCL03** | Fully implemented |
| LDRA tool suite | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | 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. |

| **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):** 8 – Practice Defense in Depth, 10 - Adopt a Secure Coding Standard  Handling all exceptions can assist in the practice of defense in depth, because these exceptions will be layered throughout the system. Including exceptions for data checking can prevent malicious data from propagating through a system. In addition, the proper use of exceptions should be standard practice and fall in line with adopting a secure coding standard. Using exceptions throughout a program can greatly reduce security vulnerabilities. |
| --- |

**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 | 2023.1 | **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** | **Honor exception specifications** |
| --- | --- | --- |
| **Exceptions** | [STD-008-CPP] | If a function declared with a *dynamic-exception-specification* throws an exception of a type that would not match the *exception-specification*, the function std::unexpected() is called. The behavior of this function can be overridden but, by default, causes an exception of std::bad\_exception to be thrown. Unless std::bad\_exception is listed in the *exception-specification*, the function std::terminate() will be called. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) noexcept(**true**) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function's *noexcept-specification* is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) {    v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):** 3 – Architect and Design for Security Policies  When exceptions are planned for a system, properly honoring specifications will ensure proper functionality. If not called properly, these exceptions will be terminated before the desired outcome. This should be considered upon the design of the overall system and when planning the flow of data through subsystems. | **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- | --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **unhandled-throw-noexcept** | Partially checked |
| CodeSonar | 8.1p0 | **LANG.STRUCT.EXCP.THROW** | Use of throw |
| LDRA tool suite | 9.7.1 | **56 D** | Partially implemented |
| Parasoft C/C++Test | 2023.1 | **CERT\_CPP-ERR55-a** | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-009-CPP | C++ does not allow you to change the value of a reference type, effectively treating all references as being const qualified. Thus, 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):** 2 – Heed Compiler Warnings, 4 – Keep it Simple  The compiler will warn developers of incorrect const qualifiers, so these warnings should be heeded to reduce the chance of this reference being ignored. Additionally, if a developer was focused on keeping the program simple, there should not be a need for const or volatile in relation to a reference type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2024.2 | **CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE** |  |
| Parasoft C/C++test | 2023.1 | **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. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Range check element access** |
| --- | --- | --- |
| **String Correctness** | [STD-010-CPP] | The std::string index operators const\_reference operator[](size\_type) const and reference operator[](size\_type) return the character stored at the specified position, pos. When pos >= size(), a reference to an object of type charT with value charT() is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the value returned by the call to get\_index() may be greater than the number of elements stored in the string, resulting in undefined behavior. |
| #include <string>    **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");    s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the std::basic\_string::at() function, which behaves in a similar fashion to the index operator[] but throws a std::out\_of\_range exception if pos >= size(). |
| #include <stdexcept>  #include <string>  **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");  **try** {      s.at(get\_index()) = '1';    } **catch** (std::out\_of\_range &) {      // Handle error    }  } |

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

| **Principles(s):** 1 – Validate Input Data  When data is introduced to a system, that data should be validated to be of the correct type and within the allowed range. Checking the range element access is a part of validating input, and the data structure being altered should also be checked to ensure no values are being stored in an index outside the bounds of that structure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun |
| Helix QAC | 2024.2 | **C++3162, C++3163, C++3164, C++3165** |  |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-STR53-a** | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2024a | CERT C++: STR53-CPP | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |

### 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.

DevSpecOps improves upon DevOps by integrating security into the SDLC from the very beginning, which aligns with the paradigm of “don’t leave security to the end”. The primary goal of DevSpecOps is to apply security principles in each phase of the SDLC, which is shown in the figure above, and emphasizes security, development, and operations equally.

For us, automation will enforce standards in several areas within the existing DevOps. Consider pre-production. A major automation factor can exist in the verify and test phase. We may implement unit testing tools such as QUnit. Through this unit testing, individual code modules will be tested prior to code integration. Furthermore, automated integration testing can be layered in to examine compliance between code modules and to also test how data moves from the database throughout the system.

In the production phase, we may automate penetration tests to attempt common attacks such as SQL injection and cross site scripting attacks with a consistent level of quality. We could release our production code in a containerized fashion to ensure the system is operating within a known and secure environment that exists abstracted from the operating system. Finally, for monitoring and detection within the production phase, automated log sniffing and log collection can be used to detect and prevent attacks. Additionally, network monitoring can identify abnormal database hits and prevent brute-force attacks through automation. By automating these processes, we can ensure coverage all the time, and issues can be addressed as they occur.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | High | P3 | L3 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | Low | Likely | Low | P9 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Unlikely | Low | P9 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Likely | Low | P9 | L2 |
| STD-009-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-010-CPP | High | Unlikely | Medium | P6 | L2 |

### 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 in rest is a concept that states data not actively in use or transit, such as on a hard disk or in a database, is encrypted. The goal of this policy is to protect data in the event of a breech, and the attacker would still require encryption keys to utilize that data. This applies to data storage within our system. |
| Encryption at flight | Encryption in flight refers to the concept of protecting data as it’s moving through the network. This policy is important as it attempts to protect data while at its most vulnerable point. Protecting data in flight is achieved by using protocols such as SSL/TLS and HTTPS. An additional measure of protection is the use of VPN where network segments need to be joined. This encryption applies to the data moving from the backend (database) and the frontend on user devices. |
| Encryption in use | Encryption in use refers to the concept of protecting data when it’s actively being utilized. This occurs after data has been queried from the database, and while the user’s application is actively interacting with the referenced data. This is accomplished using programming techniques like utilizing protected memory, managing memory correctly, and proper validation. Utilizing these safeguards limits attacking an application server in its usefulness, as you would need to breach these types of safeguards using brute force. Encryption in use ensures that data never is unsecured during any stage of use, and this applies to the system components involving frontend data manipulation and interaction. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process in which a server or application proves that you are who you say you are. By proving authentication, a user is providing the system with a way to reject or accept them based on acceptable criteria. This policy applies to any user who attempts to access sensitive data. An authentication system should be put in place as one of the first layers of defense. Generally, this is handled through a username/password pairing, but additional forms of authentication may be included like 2-factor authentication. This policy should apply to all resources to prevent unauthorized access within our system. |
| Authorization | Authorization refers to an authenticated user’s rights and privileges. Authorization is used to control a user’s level of access, most often through role-based permissions that define what a user can do. Paired with the principle of default deny, users should be denied access to all subsystems unless a specific role grants permission to complete required tasks. This policy is important to upheld as it prevents authenticated users without the proper authorization from accessing data, they should not be able to access. This applies to our scenario because access should be denied to any subsystem by default, and we can grant higher user roles if required to complete specific actions. |
| Accounting | Accounting is the process of tracking changes to a particular system or resource. Through accounting, risky behavior can be addressed early before damage is done. Accountability can be pursued on users within the system, as proof of action is available. Lastly, successful attackers can leave a trail which shows the error or exploitable portions of the system. This policy applies to our scenario because we want to protect unauthorized intrusion on our protected data. |

**\***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 | 09/15/2024 | Milestone 1 | Ryan Hubbuck |  |
| 1.2 | 10/11/2024 | Project 1 | Ryan Hubbuck |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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