**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 | This principle of security focuses on analyzing/processing data that is being input into a system such that it is rendered non-malicious, and free of error. This principle is a key focus in concepts such as SQL Injection Prevention and Buffer Overflow Prevention. In addition to denying or sanitizing input in some scenarios involving malicious attempts, input validation is also used liberally in non-malicious scenarios, such as checking or sanitizing input for correct formatting (e.g., dates, times, etc.). By validating input, we can ensure our systems are receiving only the expected input, which will mitigate a plethora of both bugs, and security concerns. |
| 1. Heed Compiler Warnings | Compiler warnings alert the developer that potential concerns exist in their code, including but not limited to deprecated code, security concerns, syntax errors, and a plethora of other issues. Compiler warnings should always be investigated, at the very least, and if left open, should be documented within the code as to why. |
| 1. Architect and Design for Security Policies | This principle is intended to ensure that security is a forethought, not an afterthought. Much like real-world building architecture, code architecture is also best constructed using a detailed, thought-out, intentional plan, rather than a security plan thrown together after the fact. Security should be intentionally intertwined and built in at the core of the program, rather than added after the program is written. |
| 1. Keep It Simple | Keeping code as simple as possible makes it easier to understand, maintain, test, and analyze. The more complex that one makes any system, the more difficult it is to secure, maintain, and otherwise edit. Increased complexity also allows for more vulnerability, generally speaking, especially when it comes to software. |
| 1. Default Deny | This principle is intended to deny all attempts to access data by default, unless access is explicitly permitted. This helps mitigate human error in design, or system bugs that would otherwise accidentally allow access where not intended. |
| 1. Adhere to the Principle of Least Privilege | This principle ties into default deny, but is specifically referring to privileges within a software environment/program. Functions, programs, users, or any other components should be compartmentalized and given only the minimum amount of required access to other compartments of the program to complete their function. This prevents unwanted intrusion whether by accident or intent. |
| 1. Sanitize Data Sent to Other Systems | Similar to input validation in nature, data sanitization ensures that data being sent to/from systems or components of a system is sanitized in order to neutralize any malicious code that may be present in the data. |
| 1. Practice Defense in Depth | Defense-in-Depth is the principle of standing up multiple layers and types of defenses. When used correctly, this principle allows users to use a system efficiently, while thwarting would-be attackers by making malicious efforts more time-consuming/difficult than their worth, should they ever be successful. By forcing attackers to implement several different types of attacks through different layers/defenses, we make it so the juice is not worth the squeeze, so-to-speak. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance is a core principle of software engineering in general, even putting security aside. Software should be thoroughly and intensely tested/vetted for all reasonably conceivable inputs/outputs, so that vulnerabilities and bugs can be discovered and addressed prior to the release of the software. |
| 1. Adopt a Secure Coding Standard | Secure coding standards have been developed because they are tried and tested against known security vulnerabilities, leading to efficient use of the developer’s time with maximum benefit to security. While secure coding standards don’t make software completely intrusion-proof, they provide an extremely strong base level of security. |

### 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** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | DCL53-CPP | Do not write syntactically ambiguous declarations. These can result in unpredictable behavior by the compiler, which is always a security concern. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an anonymous local variable of type std::unique\_lock is expected to lock and unlock the mutex m by virtue of [RAII.](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-RAII) However, the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named m and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked. |
| #include <mutex>    **static** std::mutex m;  **static** **int** shared\_resource;    **void** increment\_by\_42() {    std::unique\_lock<std::mutex>(m);    shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lock object is given an identifier (other than m) and the proper converting constructor is called. |
| #include <mutex>    **static** std::mutex m;  **static** **int** shared\_resource;    **void** increment\_by\_42() {    std::unique\_lock<std::mutex> lock(m);    shared\_resource += 42;  } |

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

| **Principles(s):** The principle of 4. Keep it Simple, applies best to this standard. Ambiguity among declarations creates overly complex-looking code. Syntactically readable code results in simpler code, which is more maintainable, and much easier to test and edit. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0po | **LANG.STRUCT.DECL.FNEST** | Nested Function Declaration |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 | **C++1109, C++2510** | None |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.4 | **CERT.DCL.AMBIGUOUS\_DECL** | None |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-DCL53-a** **CERT\_CPP-DCL53-b CERT\_CPP-DCL53-c** | Parameter names in function declarations should not be enclosed in parentheses Local variable names in variable declarations should not be enclosed in parentheses Avoid function declarations that are syntactically ambiguous |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT32-C | Ensure that operations on signed integers do not result in overflow. Overflow can result in unpredictable behavior including security compromise, allowing malicious code to operate, or unauthorized access. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in a signed integer overflow during the addition of the signed operands si\_a and si\_b: |
| **void** func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the addition operation cannot overflow, regardless of representation: |
| #include <limits.h>    **void** f(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum;  **if** (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||        ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {      /\* Handle error \*/    } **else** {      sum = si\_a + si\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** Heeding compiler warnings would be the principle which best maps to this rule/standard. Most (not all) IDEs will throw a compiler warning upon detecting an impending potential buffer overflow, and this becomes even more assured when working with analyzers such as cppcheck. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **integer-overflow** | Fully checked. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **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](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **TAINTED\_SCALAR**  **BAD\_SHIFT** | Implemented |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.11.0 | **premium-cert-int32-c** | None |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR31-C | Guarantee that storage for strings has sufficient space for character data and the null terminator. Improper string handling can cause buffer overflows, memory leaks/corruption, and unauthorized access to a system. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates an *off-by-one* error [[Dowd 2006](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Dowd06)]. The loop copies data from src to dest. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of dest. |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to dest: |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n - 1); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |

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

| **Principles(s):** The principle of 1. Validate Input Data would best map to this rule. Buffer overflow is preventable using input validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 |  | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR31** | Detects calls to unsafe string function that may cause buffer overflow Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **LANG.MEM.BO LANG.MEM.TO MISC.MEM.NTERM BADFUNC.BO.\*** | Buffer overrun Type overrun No space for null terminator A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| [Compass/ROSE](https://www.securecoding.cert.org/confluence/display/seccode/Rose) |  |  | Can detect violations of the rule. However, it is unable to handle cases involving strcpy\_s() or manual string copies such as the one in the first example |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | FIO30-C | Exclude user input from format strings. This is the core principle of SQL Injection Prevention, as user input in format strings is the primary means of SQL Injection. Sanitizing and/or otherwise preventing direct user input from format strings is the first line of defense against SQL Injection. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of [untrusted data](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-untrusteddata) that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **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 \*/    }  **fprintf**(stderr, msg);  **free**(msg);  } |

| **Compliant Code** |
| --- |
| This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents: |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **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);  } |

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

| **Principles(s):** The principle of 1. Validate Input Data would best map to this rule. Sanitizing input data is a form of input validation, and that’s exactly what this rule addresses. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 |  | Supported via stubbing/taint analysis |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-FIO30** | Partially implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **IO.INJ.FMT MISC.FMT** | Format string injection Format string |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **TAINTED\_STRING** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM30-C | Do not access freed memory. Allowing a program to access freed, or deallocated memory creates risk of unpredictable behavior and/or malicious code execution. |

| **Noncompliant Code** |
| --- |
| This example from Brian Kernighan and Dennis Ritchie [[Kernighan 1988](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Kernighan88)] shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, p is freed before p->next is executed, so that p->next reads memory that has already been freed. |
| #include <stdlib.h>    **struct** node {  **int** value;  **struct** node \*next;  };    **void** free\_list(**struct** node \*head) {  **for** (**struct** node \*p = head; p != NULL; p = p->next) {  **free**(p);    }  } |

| **Compliant Code** |
| --- |
| Kernighan and Ritchie correct this error by storing a reference to p->next  in q before freeing p: |
| #include <stdlib.h>    **struct** node {  **int** value;  **struct** node \*next;  };    **void** free\_list(**struct** node \*head) {  **struct** node \*q;  **for** (**struct** node \*p = head; p != NULL; p = q) {      q = p->next;  **free**(p);    }  } |

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

| **Principles(s):** 9. Use Effective Quality Assurance Techniques would map best to this rule. Analysis tools such as Cppcheck, and unit testing would likely be the best tools to catch instances of freed memory usage, as they are often not readily apparent to a developer, and easily missed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **dangling\_pointer\_use** | Supported  Astrée reports all accesses to freed allocated memory. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM30** | Detects memory accesses after its deallocation and double memory deallocations |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **ALLOC.UAF** | Use after free |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/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 |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | ERR51-CPP | Handle all exceptions. Unhandled exceptions can cause system crashes, memory leaks, and other unpredictable behavior depending on the exception, compromising security and program integrity. |

| **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 would best map to this rule. It’s a bit broad, but so are exceptions- and it’s important to throw and handle exceptions for undefined behaviors, or unexpected inputs, as we can’t prevent every last situation possible- exceptions account for the unexpected, or the “exception” to the rule, thusly named. In the case of assertions, an exception is thrown if an assertion returns false, and that must be handled appropriately, even though assertions are intended to always return true. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR51** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
|  | [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR50-CPP | Do not abruptly terminate the program. Incorrectly terminating a program could potentially create holes in security, compromise system performance, and allow unauthorized access to data. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(), may result in a call to std::terminate() because throwing\_func() may throw an exception. |
| **#include <cstdlib>**    **void throwing\_func() noexcept(false);**    **void f() { // Not invoked by the program except as an exit handler.**  **throwing\_func();**  **}**    **int main() {**  **if (0 != std::atexit(f)) {**  **// Handle error**  **}**  **// ...**  **}** |

| **Compliant Code** |
| --- |
| In this compliant solution, f() handles all exceptions thrown by throwing\_func() and does not rethrow. |
| **#include <cstdlib>**    **void throwing\_func() noexcept(false);**    **void f() { // Not invoked by the program except as an exit handler.**  **try {**  **throwing\_func();**  **} catch (...) {**  **// Handle error**  **}**  **}**    **int main() {**  **if (0 != std::atexit(f)) {**  **// Handle error**  **}**  **// ...**  **}** |

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

| **Principles(s):** 8. Practice Defense in Depth would best map to this rule. It’s a bit broad, but so are exceptions- and it’s important to throw and handle exceptions for undefined behaviors, or unexpected inputs, as we can’t prevent every last situation possible- exceptions account for the unexpected, or the “exception” to the rule, thusly named. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **stdlib-use** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **BADFUNC.ABORT BADFUNC.EXIT** | Use of abort Use of exit |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 | **C++5014** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.4 | **MISRA.TERMINATE** **CERT.ERR.ABRUPT\_TERM** |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations | DCL60-CPP | Obey the one-definition rule. Failure to obey the one-definition rule can lead to unpredictable system behavior in many cases at worst, and at best, will lead to confusion when maintaining a code base. |

| **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. This code example violates the ODR and results in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |
| // 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):** 4. Keep it Simple best maps to this rule, as creating several different definitions creates confusion in code by increasing complexity, and can lead to unpredictable system behavior both directly, an indirectly (future programmers maintaining code may accidentally map function/class/struct calls to incorrect definitions). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **type-compatibility definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading type-file-spreading** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL60** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **LANG.STRUCT.DEF.FDH LANG.STRUCT.DEF.ODH** | Function defined in header file Object defined in header file |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 | **C++1067, C++1509, C++1510** |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations | DCL58-CPP | Do not modify the standard namespaces. Standardized namespaces are heavily vetted for security principles, and are standardized to behave in a specific, predictable manner. Modifying standardized namespaces, or other standardized architectures in software can result in highly unpredictable, sometimes catastrophic behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the declaration of x is added to the namespace std, resulting in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |
| **namespace** std {  **int** x;  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes the intention of the programmer was to place the declaration of x into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name. |
| **namespace** nonstd {  **int** x;  } |

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

| **Principles(s):** 3. Architect and Design for Security Policies best maps to this rule, as standardized namespaces are designed/published under heavy scrutiny with quality assurance and unit testing thoroughly conducted before release. Modifying these namespaces is akin to modifying a vehicle and breaking the warranty. The security guarantees are made only through proper use of the already vetted namespace remaining intact. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL58** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **LANG.STRUCT.DECL.SNM** | Modification of Standard Namespaces |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 | **C++3180, C++3181, C++3182** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.4 | **CERT.DCL.STD\_NS\_MODIFIED** |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | FIO47-C | Use valid string formats. Incorrect string formatting can result in undefined behavior, or even allow system intrusion via SQL Injection or unintended memory access. |

| **Noncompliant Code** |
| --- |
| Mismatches between arguments and conversion specifications may result in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-undefinedbehavior). Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the error\_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. Likewise, the error\_msg argument is incorrectly matched with the d specifier instead of the s specifier. These usages result in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-undefinedbehavior). One possible result of this invocation is that printf() will interpret the error\_type argument as a pointer and try to read a string from the address that error\_type contains, possibly resulting in an access violation. |
| #include <stdio.h>    **void** func(**void**) {  **const** **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;    /\* ... \*/  **printf**("Error (type %s): %d\n", error\_type, error\_msg);    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications: |
| #include <stdio.h>    **void** func(**void**) {  **const** **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;    /\* ... \*/  **printf**("Error (type %d): %s\n", error\_type, error\_msg);      /\* ... \*/  } |

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

| **Principles(s):** 1. Validate Input Data maps quite directly to this rule, as invalid string inputs can cause unpredictable behavior and results in buffer overflow and SQL injection in many cases. Invalid inputs are preventable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-FIO47** | Fully implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **IO.INJ.FMT MISC.FMT MISC.FMTTYPE** | Format string injection Format string Format string type error |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **PW** | Reports when the number of arguments differs from the number of required arguments according to the format string |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 2.15 | **invalidscanf** **wrongPrintfScanfArgNum** **invalidLengthModifierError** **invalidScanfFormatWidth** **wrongPrintfScanfParameterPositionError** |  |

### 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 | High | Unlikely | Medium | High | 2 |
| DCL53-CPP | Low | Unlikely | Medium | P2 | L3 |
| DCL58-CPP | High | Unlikely | Medium | **P6** | **L2** |
| DCL60-CPP | High | Unlikely | High | **P3** | **L3** |
| ERR50-CPP | Low | Probable | Medium | **P4** | **L3** |
| ERR51-CPP | Low | Probable | Medium | **P4** | **L3** |
| FIO30-C | High | Likely | Medium | **P18** | **L1** |
| FIO47-C | High | Unlikely | Medium | **P6** | **L2** |
| INT32-C | High | Likely | High | **P9** | **L2** |
| MEM30-C | High | Likely | Medium | **P18** | **L1** |
| STR31-C | High | Likely | Medium | **P18** | **L1** |

### 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 | **What it is:** Encryption at rest provides protection for data that is at rest (being stored in memory, typically this refers to nonvolatile memory, but not always). Encryption is performed by using a key to scramble/rearrange the binary of an input in order to produce a different output that can be reversed using a key (in some cases the key will be the same, such as XOR encryption, in other cases the key will be different, such as RSA encryption. This is uncommon for data at rest, but not explicitly unused) to produce the original input.  **Policy:**  **How it should be applied in practice: All sensitive data stored in databases shall be stored using an AES-256 Data Encryption Key (DEK), and using the principle of least privilege, shall only be accessible by users that explicitly require access to the corresponding data. Each user shall have their own specific KEK (Key Encryption Key), such that the user cannot accidentally compromise the DEK itself.**  **Why it should be used: In the case that the user’s KEK is compromised, this will provide the maximum amount of encapsulation of data by restricting unnecessary access, while allowing unhindered access to data by those who explicitly require it, and possibly allowing the DEK to remain uncompromised for the duration of a data breach (though it shall be regenerated in this case regardless). These protocols will add several layers of security just using one method- encryption.** |
| Encryption in flight | **What it is:** Encryption in flight, AKA Encryption in transit, provides protection for data that is actively traveling over any network. While similar to encryption at rest, it is unique such that the sender must be able to encrypt the data, and allow the recipient to remotely decrypt the data while simultaneously not transmitting the method which allows the data to be decrypted. This is where RSA encryption shines.  **Policy:**  **How it should be applied in practice: All users will use the Green Pace VPN before being allowed to log into the system. This will ensure that all traffic is encrypted from the origin prior to being transmitted over the internet. Green Pace VPN uses logging such that all traffic at the mainframe endpoint is logged into nonvolatile memory, so any malicious behavior can be analyzed. Similarly, our web server endpoints shall all use https protocols with certificate validation.**  **Why it should be used: Encryption in transit needs to be used in order to protect sensitive data while being sent from any sender to any recipient. In addition to preventing the release of data, it also prevents the manipulation of data such as that of a man-in-the-middle attack. Using a VPN in combination with https protocols is the simplest, most cost-effective form of providing encryption in flight for our company.** |
| Encryption in use | **What it is:** Encryption in use encrypts data while in a state of use, which can be loosely defined as any volatile state not on a network (that would move into the definition of “in transit”).  **Policy:**  **How it should be applied in practice: All VM instances for remote work users shall have AMD SEV (AMD Secure Encrypted Virtualization) enabled to protect proprietary company and client data in addition to VPN use.**  **Why it should be used: As a failsafe in case any other security measure fails. In the event of a buffer overflow, or any other possible volatile memory breach, the data will be unusable by unauthorized parties.** |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication: Manages access to a system using identity verification in one or more methods.  **Policy:**  **How: All users shall be required to maintain a physical access keycard with an individualized PIN (along with a card reader) in addition to their username and password. Passwords shall be changed once per year, consisting of 8-16 characters, including two lowercase, two uppercase, two integer, and two special characters at a minimum. Passwords shall be salted in the database.**  **Why: This triple authentication method requires three forms of identity verification with a dual-medium requirement (one physical, two digital). Requiring a keycard with a pin creates a simple, easy to use, yet extremely secure entry barrier for user authentication, while a username and password creates an extra layer of security, with a one-year password reset requirement allowing the user to remember a unique password without the temptation to write it down due to frequent resets. Password salting is a best practice for username/password implementation.** |
| Authorization | Authorization: Compartmentalizes access to a system using authentication methods such that only specific entities are allowed access to specific compartments/modules of a system.  **Policy:**  **How: All users shall receive specific access to each section of the company’s file system explicitly as defined by their job requirements upon initialization of new users. Access will be granted and removed by administrators as each project or assignment requires. The request system for access shall be user-friendly, and will be a top priority for administrators. All access requests shall be approved or denied within 24 hours of receipt.**  **Why: Limiting system access using the principle of least privilege ensures that in the inevitable event of a data breach, the breach is limited only to the data accessible by the compromised user account. This compartmentalization is extremely useful at creating a barrier to entry for any unauthorized parties attempting to gain access to systems.** |
| Accounting | Accounting is the practice of logging the flow of data in/out of a system, along with usage of that data (all data in transit, and in use, as applicable per policy).  **Policy:**  **How: All VPN and internal system traffic shall be permanently stored on company drives using a RAID 6 Array. These drives will be removed, stored in the company SCIF (we work with government contracts, and therefore we require a SCIF), and replaced when full on a randomized basis at a minimum of twice per month in order to create an air gap between any attacks and the stored audit trails. These hard drives shall only be analyzed on standalone machines with no network connection whatsoever.**  **Why: Log keeping allows root cause analysis to take place on malicious events, even when discovered long after the fact. A RAID 6 Array specifically allows for dual redundancy, while allowing permanent storage on cost effective, stable HDDs. The removal and storage of these HDDs as they fill creates an air gap as stated, which supports the Defense-in-Depth concept, preventing tampering or manipulation.** |

**\***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 | Brad Emerson |
| 1.1 | 4/11/2025 | Initial Completed Draft | Brad Emerson |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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