**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 12](#_Toc52464067)

[Coding Standard 10 13](#_Toc52464068)

[Defense-in-Depth Illustration 14](#_Toc52464069)

[Project One 14](#_Toc52464070)

[1. Revise the C/C++ Standards 14](#_Toc52464071)

[2. Risk Assessment 14](#_Toc52464072)

[3. Automated Detection 14](#_Toc52464073)

[4. Automation 14](#_Toc52464074)

[5. Summary of Risk Assessments 15](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 15](#_Toc52464076)

[7. Map the Principles 16](#_Toc52464077)

[Audit Controls and Management 17](#_Toc52464078)

[Enforcement 17](#_Toc52464079)

[Exceptions Process 17](#_Toc52464080)

[Distribution 18](#_Toc52464081)

[Policy Change Control 18](#_Toc52464082)

[Policy Version History 18](#_Toc52464083)

[Appendix A Lookups 18](#_Toc52464084)

[Approved C/C++ Language Acronyms 18](#_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. Validate Input Data | This is the principle that ensures all input that is received from an external source (including user inputs, website requests, and APIs) is safe to be used and in the correct format before being processed by the system. Validating input data protects the system from vulnerabilities like injection attacks, and data corruption and prevents applications from crashing. |
| 1. Heed Compiler Warnings | This principle highlights the importance of paying attention to any warning output by the compiler while testing and running code. The compiler can issue a wide variety of errors that can help developers identify bugs in the code that could compromise the program when trying to run it. Ignoring compiler warnings can lead to unstable and insecure code that does not run correctly or does not run at all. |
| 1. Architect and Design for Security Policies | The principle of building security standards into the program’s architecture throughout the software development process. This ensures that potential vulnerabilities are considered at the very first stages of writing code. This makes it easier to create robust security for the system that will function well at all levels. |
| 1. Keep It Simple | Keeping it simple means writing code and corresponding security standards that are not overly complex since this makes it more difficult to maintain the system. A complicated system is more difficult to maintain and harder to teach others to maintain. This invites the chance for vulnerabilities to be overlooked and exploited. |
| 1. Default Deny | This principle means to deny access to resources by default and only allow access when explicitly allowed to do so. This prevents a wide variety of vulnerabilities by letting the user know when something would like to access the system and giving them a choice to allow or deny access. |
| 1. Adhere to the Principle of Least Privilege | This principle grants the users and applications the minimum level of access necessary to fulfill intended tasks. This prevents the potential misuse of privileges and compromising the system and its security. |
| 1. Sanitize Data Sent to Other Systems | This principle prioritizes that data sent from one system to another is clean and properly formatted. This helps to ensure that the data is free of malicious intent or external threats. This is particularly helpful against injection attacks or other attacks that originate from outside of the system. |
| 1. Practice Defense in Depth | This principle involves implementing multiple layers of security controls and defenses to protect systems and data. This multi-layered approach ensures that if one layer fails or is breached, other layers continue to provide protection. |
| 1. Use Effective Quality Assurance Techniques | This principle means to use rigorous testing and quality assurance strategies to keep code reliable and secure. This principle can be employed at almost every stage of the software development lifecycle as it helps identify potential security issues and vulnerabilities. |
| 1. Adopt a Secure Coding Standard | This principle involves establishing and following a set of coding guidelines designed to minimize security vulnerabilities and promote the development of secure software. This principle ensures that all developers within an organization adhere to best practices that enhance the security, reliability, and maintainability of the code |

### 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** | DCL-51-CPP | Do not declare or define a reserved identifier. |

| **Noncompliant Code** |
| --- |
| This example uses a trailing underscore to define MY\_HEADER. This can confuse the program since some underscored titles are reserved as headergueards. |
| #ifndef \_MY\_HEADER\_H\_  #define \_MY\_HEADER\_H\_    // Contents of <my\_header.h>    #endif // \_MY\_HEADER\_H\_ |

| **Compliant Code** |
| --- |
| This example does not use a trailing underscore and avoids confusing the program between the contents of the code block and the header guards. |
| #ifndef MY\_HEADER\_H  #define MY\_HEADER\_H    // Contents of <my\_header.h>    #endif // MY\_HEADER\_H |

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

| **Principles(s):**  **Keeping it Simple:** Writing simple code, in both logic and syntax makes it less likely to encounter errors or confuse the compiler and user. Removing the extra ‘\_’ at the end of the header makes it easier to type, remember, and prevents the header from being mistaken for another kind of function that requires an underscore at the end.  **Heed compiler Warnings:** The compiler may generate a warning if this header has potential to be recognized as something it is not. Just because the program runs without crashing does not mean that such a warning should be ignored. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Reserved-identifier | Partially checked |
| Axivision | 7.2.0 | CertC++ -DCL51 | - |
| Clang | 3.9 | -Wreserved-id-macro -Wuser-defined-literals | The -Wreserved-id-macro flag is not enabled by default or with -Wall, but is enabled with -Weverything. This flag does not catch all instances of this rule, such as redefining reserved names. |
| Code Sonar | 8.1p0 | LANG.ID.NU.MK  LANG.STRUCT.DECL.RESERVED | Macro name is C keyword  Declaration of reserved name |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT32-C | Ensure that operations on signed integers do not result in overflow |

| **Noncompliant Code** |
| --- |
| If represented by the right values, si\_a and si\_b, when added together will result in overflow. |
| **void** func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This code ensures that si\_a and si\_b do not result in overflow by adding conditional operators to the operation. |
| #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):**  **Default Deny:** Parameters can be put in place to prevent overflow of signed integer operations. Buffer overflow would allow a range of values to be established after operations are complete. Depending on the code function, the user can put logic in place to prevent operations that exceed a certain range or provide a notation that shows how the value might exceed the parameters.  **Validate Input Data:** Ensure that the values of signed integers are appropriate for the operations to come. This can prevent logical errors that end with values outside of a specified range or data type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 34.04 | Integer-overflow | Partially checked |
| CodeSonar | 8.1p0 | **ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | implemented |
| Helix QAC | 2024.2 | **C2800, C2860**  **C++2800, C++2860**  **DF2801, DF2802, DF2803, DF2861, DF2862, DF2863** | - |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR30-C | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| In this example, the char pointer is initialized to the string literal. Modifying a string literal is considered undefined behavior and may result in an error. |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| This code is compliant by defining the string literal in a vector. Pieces of the string literal can then be redefined by calling different indices of the vector and assigning them a new value. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):**  **Validate Input Data:** Attempting to alter string literals usually results in some form of syntax error, but string literals usually exist for good reasons. String literals often indicate direction or instruction for developers and users. Altering them may result in new bugs and errors. Making sure that string liters are all denoted with the proper quotations is helpful before trying to make changes to anything else. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | String-literal-modification  Write-to-string-literal | Fully Checked |
| Axivision Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Compass/ROSE | - | - | Can detect simple violations of this rule |
| Helix QACV | 2024.2 | **C0556, C0752, C0753, C0754**  **C++3063, C++3064, C++3605, C++3606, C++3607** | - |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | MEM51-CPP | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| The local carriable ‘space’ is passed as the expression to the placement ‘new’ operator. The pointer is then called and passed to ::operator\_delete(), resulting in undefined behavior. This behavior is similar to how attackers can overload inputs and gain access to restricted data. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...    **delete** s1;  } |

| **Compliant Code** |
| --- |
| This code removes the call to ::operator\_delete() and calls ‘s1’ destructor. This prevents the use of potential overaload. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...      s1->~S();  } |

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

| **Principles(s):**  **Adhere to the Principle of Least Privilege:** Deallocating resources as needed prevents end users from having authorization or access they do not need. When allocated, the resource performs its designated function, which may give it specific access to interact with another function or resource. Deallocating it before anything else happens means denying the user any access that does not have anything to do with the way they interact with the program.  **Practice Defense in Depth:** Practicing defense in depth means having security measures in place to make sure resources are deallocated before any other code is compiled. This can exist in the form of an exception that is thrown that alerts the developer that a resource was not deallocated before moving on to other functions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | **invalid\_dynamic\_memory\_allocation dangling\_pointer\_use** | - |
| Axivision Bauhaus Suite | 7.2.0 | CERTC++ -MEM51 | - |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 8.1p0 | **ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM50-CPP | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| Variable ‘s’ is dereferenced after it has been deallocated. Dynamic memory allocations are usually not considered while integrating security standards and can often be overlooked. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant example, the memory of variable ‘s’ is not deallocated until it is not needed anymore |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |

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

| **Principles(s):**  **Adopt a Secure Coding Standard:** Maintaining the standard of not accessing freed memory may help programs run smoother. Depending on the program and the hardware it is running on. Free memory may be necessary for other functions. Limited RAM may also prevent programs from functioning correctly. Ensuring that there is enough memory for specified programs is a good standard to set. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 L1 | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Dangling\_pointer\_use | - |
| Axivison Bauhaus Suite | 7.2.0 | CERTC++ -MEM50 | - |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| Klocwork | [Insert text.] | **UFM.DEREF.MIGHT** **UFM.DEREF.MUST** **UFM.FFM.MIGHT** **UFM.FFM.MUST** **UFM.RETURN.MIGHT** **UFM.RETURN.MUST** **UFM.USE.MIGHT** **UFM.USE.MUST** | - |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | ERR57-CPP | Do not leak resources when handling exceptions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, pst is not properly released when process\_item throws an exception, causing a resource leak. |
| #include <new>    **struct** SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.  **void** process\_item() noexcept(**false**);  };    **void** f() {    SomeType \*pst = **new** (std::**nothrow**) SomeType();  **if** (!pst) {      // Handle error  **return**;    }    **try** {      pst->process\_item();    } **catch** (...) {      // Process error, but do not recover from it; rethrow.  **throw**;    }  **delete** pst;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the exception handler frees pst by calling delete. This prevents issues like overflow or errors when assigning new data to variables by clearing previous data and declarations. |
| #include <new>    **struct** SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.    **void** process\_item() noexcept(**false**);  };    **void** f() {    SomeType \*pst = **new** (std::**nothrow**) SomeType();  **if** (!pst) {      // Handle error  **return**;    }  **try** {      pst->process\_item();    } **catch** (...) {      // Process error, but do not recover from it; rethrow.  **delete** pst;  **throw**;    }  **delete** pst;  } |

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

| **Principles(s):**  **Architect and Design for Security Policies:** Ensure that resources are deallocated and local variables are reset (if necessary) when handling exceptions. Some exceptions force the program to crash to protect data. If such an exception is not in place, then data is at risk of being leaked because the program never crashed. Try-catch logic is helpful in the scenario because it can look for a wider variety of issues.  **Use Effective Quality Assurance Techniques:** Utilizing try-catch over logic like if-else is helpful because it has a broader scope of scenarios it can look out for. If-else logic generally does not throw an exception like try-catch can. This is important when dealing with sensitive data that cannot afford to be leaked. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2024.2 | **DF4756, DF4757, DF4758** | - |
| KlocWork | 2024.2 | **CL.MLK** **MLK.MIGHT** **MLK.MUST** **MLK.RET.MIGHT** **MLK.RET.MUST** **RH.LEAK** | - |
| LDRA tool suite | 9.7. | 50 D | Partially implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR50-CPP | Do not abruptly terminate the program. |

| **Noncompliant Code** |
| --- |
| This code may result in an unexpected termination of the program because std::at\_exit() may result in calling std::terminate() because an exception may be thrown. |
| #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 example, f() handles exceptions thrown by throwing\_func(). It does not rethrow any exceptions, avoiding the risk of termination of the program. |
| #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):**  **Use Effective Quality Assurance Techniques:** Terminating a program abruptly can have consequences on the program itself. Data from cache memory may corrupt or be lost completely if there is not something in place to handle an unexpected termination.  **Architect and Design for Security Policies:** Designing code to handle an unexpected shutdown can protect against data leaks. Having a function in place that dumps cache memory or deallocates local resources can keep data secure that was supposed to stay within the program's functions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stdlib-use | Partially checked |
| CodeSonar | 8.1p0 | **BADFUNC.ABORT BADFUNC.EXIT** | Use of abort Use of exit |
| Helix QAC | 2024.2 | C++ 5014 | - |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-ERR50-a CERT\_CPP-ERR50-b CERT\_CPP-ERR50-c CERT\_CPP-ERR50-d CERT\_CPP-ERR50-e CERT\_CPP-ERR50-f CERT\_CPP-ERR50-g CERT\_CPP-ERR50-h CERT\_CPP-ERR50-i CERT\_CPP-ERR50-j CERT\_CPP-ERR50-k CERT\_CPP-ERR50-l CERT\_CPP-ERR50-m CERT\_CPP-ERR50-n** | The execution of a function registered with 'std::atexit()' or 'std::at\_quick\_exit()' should not exit via an exception Never allow an exception to be thrown from a destructor, deallocation, and swap Do not throw from within destructor There should be at least one exception handler to catch all otherwise unhandled exceptions An empty throw (throw;) shall only be used in the compound statement of a catch handler Exceptions shall be raised only after start-up and before termination of the program 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 Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) Function called in global or namespace scope shall not throw unhandled exceptions Always catch exceptions Properly define exit handlers The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used Avoid throwing exceptions from functions that are declared not to throw The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | CON51-CPP | Ensure actively held locks are released on exceptional conditions. |

| **Noncompliant Code** |
| --- |
| This code protects critical data by locking the mutex, but if an exception occurs then it stays locked. |
| #include <mutex>    **void** manipulate\_shared\_data(std::mutex &pm) {    pm.lock();      // Perform work on shared data.      pm.unlock();  } |

| **Compliant Code** |
| --- |
| This version of the code catches any thrown exceptions, preventing the mutex from being permanently locked. |
| #include <mutex>    **void** manipulate\_shared\_data(std::mutex &pm) {    pm.lock();  **try** {      // Perform work on shared data.    } **catch** (...) {      pm.unlock();  **throw**;    }    pm.unlock(); // in case no exceptions occur  } |

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

| **Principles(s):**  **Practice Defense in Depth:** Practicing defense in depth in this scenario means that there is code in place that prevents locks from being released until specific criteria are met. This means that security has been considered on a deeper level and was integrated during the development lifecycle.  **Default Deny:** This expressly prevents any operations or functions from running on specific pieces of code unless expressly permitted by a given exception.The nice part about Default Deny is that it prevents the mistreatment of data when it should not be altered by not allowing any alteration unless expressly permitted given a predetermined circumstance. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 8.1p0 | CONCURRENCY.LOCL.NOUNLOCK | Missing Lock Release |
| Helix QAC | 2024.2 | C++ 5018 | - |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CON51-a | Do not call lock() directly on a mutex |
| Polyspace Bug Finder | R2024a | CERT C++: CON51-CPP | Checks for lock possibly not released on exception (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | FIO38-C | Do not copy a FILE object |

| **Noncompliant Code** |
| --- |
| This code might fail because it uses a by-value copy of stdout to call fouts(). Copying certain values can affect how the data behaves in different parts of the code. |
| #include <stdio.h>    **int** main(**void**) {  **FILE** my\_stdout = \*stdout;  **if** (**fputs**("Hello, World!\n", &my\_stdout) == EOF) {      /\* Handle error \*/    }  **return** 0;  } |

| **Compliant Code** |
| --- |
| This code uses a pointer in \*my\_stdout to point to the FILE object instead of using a copy |
| **int** main(**void**) {  **FILE** \*my\_stdout = stdout;  **if** (**fputs**("Hello, World!\n", my\_stdout) == EOF) {      /\* Handle error \*/    }  **return** 0;  } |

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

| **Principles(s):**  **Validate Input Data:** It may be good practice to keep FILE objects from being copied to prevent an error.  **Keep It Simple:** It may be easier to maintain the values of FILE objects by saving them to variables rather than calling them directly. Depending on the program, these objects can cause unexpected errors and complicate the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.5 | Partially implemented |
| Helix QAC | 2024.2 | **C1485, C5028**  **C++3113, C++3114** | - |
| Klocwork | 2024.2 | **MISRA.FILE\_PTR.DEREF.2012** **MISRA.FILE\_PTR.DEREF.CAST.2012** **MISRA.FILE\_PTR.DEREF.INDIRECT.2012** **MISRA.FILE\_PTR.DEREF.RETURN.2012** | - |
| LDRA tool suite | 9.7.1 | 591 S | Fully implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | CTR53-CPP | Use valid iterator ranges. |

| **Noncompliant Code** |
| --- |
| The two iterators marking the range are within the same container, but the first iterator does not come before the second. During each iteration of its internal loop, std::for\_each() increments the first iterator and checks if it equals the second; since this process continues, it eventually leads to undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this code, the iterators are passed in the correct order, preventing undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):**  **Use Effective Quality Assurance Techniques:** Using valid iterator ranges provides the code with logic on what to consider as valid output. If logic needs to follow a specified output based on the range of an iterator, then that range needs to be declared before any functions follow. This could mean specifying the data types that will be iterated through or a range of numbers depending on the operators. Declaring these things as early as possible prevents undefined behavior and potential errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Overflow\_upon\_dereference | - |
| CodeSonar | 8.1p0 | LANG\_MEM.BO | Buffer Overrun |
| Helix QAC | 2024.2 | C++3802 | - |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-CTR53-a** **CERT\_CPP-CTR53-b** | Do not use an iterator range that isn't really a range Do not compare iterators from different containers |

### 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.

Automating the standards of the DevOps process involves the integration of tools like Astree, Helix QZC, CodeSonar, Parasoft C/C++test, etc. A crucial step is to automate the static code analysis tools to allow code to be tested incrementally throughout the DevOps cycle. This keeps security a priority throughout the cycle rather than saving it for the end of the process. This process also complies with defense-in-depth by integrating security on multiple levels. Astree is a versatile tool that can check for a variety of high-risk vulnerabilities, validate input, and optimize headers. Many of these vulnerabilities are simple to fix but easy to overlook, so automating this process takes a lot of human error out of the equation.

These tools also help with the transition from pre-production to production by continuously monitoring vulnerabilities. If tools like Astree, CodeSonar, etc. previously found vulnerabilities, then they will be easier to spot and fix with each cycle. Automating these processes also allows for more attention to be brought to finding previously unchecked vulnerabilities.

### 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 |
| INT32-C | High | Likely | High | P9 | 2 |
| MEM50-CPP | High | Likely | Medium | P18 | 1 |
| MEM51-CPP | High | Likely | Medium | P18 | 1 |
| CTR53-CPP | High | Probable | High | P6 | 2 |
| DCL-51-CPP | Low | Unlikely | Low | P3 | 3 |
| STR30-C | Low | Likely | Low | P9 | 2 |
| ERR57-CPP | Low | Probable | High | P2 | 3 |
| ERR50-CPP | Low | Probable | Medium | P4 | 3 |
| CON51-CPP | Low | Probable | Low | P6 | 2 |
| FIO38-C | Low | Probable | Medium | P4 | 3 |

### 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 | Encrypting data that is stored on a disk, whether in a database, file system, or other storage method. This protects the data when it is not being actively used. The encryption key must be stored separately from the data, and that location is rotated using a key management service. This protects against unauthorized access, complies with encryption regulations, and mitigates the damage of potential data breaches. |
| Encryption in flight | Encrypting data while it is being transferred from one location to another. Data is encrypted before it is transmitted over the network. This can be done using protocols like TLS (Transport Layer Security) or SSL (Secure Sockets Layer), which secure the communication channels. This protects against data interception and Man-in-the-Middle attacks. |
| Encryption in use | Encrypting data while it is being processed or actively used. This ensures that sensitive data is protected while in use. Data is encrypted while it is in memory (RAM) and decrypted only within the CPU, ensuring that it remains protected during active use. This protects data while it is most vulnerable. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of verifying and identifying a user before accessing sensitive data. This involves user logins, possibly a multiple-step login process, recording login attempts, and preventing sign-in after too many failed attempts. This is valuable to the policy by preventing barrage-based attacks by programs that use several permutations of login credentials in a short period of time. |
| Authorization | This process determines what a user is allowed to do and what permission they have regarding data access and mobility. This involves limited database manipulation to only authorized users, adding new users, and accessing sensitive files. The policy will utilize common security practices by limiting user access only to what they need to fulfill their tasks. |
| Accounting | The tracking and recording of user activities. This helps in monitoring user actions, detecting potential security breaches, and maintaining a historical record for analysis and compliance purposes. The policy will allow authorized users to track other user activity in the event of a data breach by viewing login attempts and requests to access specific files. Being able to map out the activity of users, especially when investigating suspicious activity, is crucial to protecting 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 | 07/15/2024 | 3-2 Milestone | Austin Siegel | Austin Siegel |
| 1.2 | 08/08/2024 | Project One: Completed | Austin Siegel | Austin Siegel |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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