**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 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 22](#_Toc52464067)

[Coding Standard 10 25](#_Toc52464068)

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

[Project One 27](#_Toc52464070)

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

[2. Risk Assessment 27](#_Toc52464072)

[3. Automated Detection 27](#_Toc52464073)

[4. Automation 27](#_Toc52464074)

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

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

[7. Map the Principles 29](#_Toc52464077)

[Audit Controls and Management 30](#_Toc52464078)

[Enforcement 30](#_Toc52464079)

[Exceptions Process 30](#_Toc52464080)

[Distribution 31](#_Toc52464081)

[Policy Change Control 31](#_Toc52464082)

[Policy Version History 31](#_Toc52464083)

[Appendix A Lookups 31](#_Toc52464084)

[Approved C/C++ Language Acronyms 31](#_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 | We need to ensure that we are validating all data that is input and checking to be sure none of in data appears suspicious or potentially malicious. |
| 1. Heed Compiler Warnings | Anytime we are presented with Compiler Warnings we should not over look them. Read the warning completely and if you are unsure, reach out to someone to gain further clarification to see if it may cause issues later on. |
| 1. Architect and Design for Security Policies | Always keep security policies in mind while designing and architecting systems. With this topic at the forefront of our thoughts we can design around the security principals we need in place. |
| 1. Keep It Simple | Simple code is effective. When it gets over complicated then more errors can be introduced and can make it harder to debug. |
| 1. Default Deny | Any traffic to our network that isn’t specifically and clearly authorized should be denied access by default. This is best practice for keeping data secured. |
| 1. Adhere to the Principle of Least Privilege | We shouldn’t be granting any additional privileges to users, entities, or organizations outside of what is strictly necessary. |
| 1. Sanitize Data Sent to Other Systems | If we are pushing data to another system we have to ensure it won’t get flagged as potential spam, or suspicious data so we have to cleanse our data of any potential characters that could get it marked as such. |
| 1. Practice Defense in Depth | Layer multiple levels of security practices on top of each other so that our systems are protected from a variety of threats across the board. |
| 1. Use Effective Quality Assurance Techniques | Testing thoroughly is key to our success, by ensuring we have a high quality of code and security we can be sure we keep our systems safe from external threats. |
| 1. Adopt a Secure Coding Standard | We have to set our own standard for writing code. If we can set our own standards for the quality of work we want to produce then we can keep ourselves, each other, and all members of the team accountable to our standards for 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** | INT30-C | Ensure that unsigned integer operations do not wrap. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap: |
| #include <limits.h>    **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum;  **if** (UINT\_MAX - ui\_a < ui\_b) {      /\* Handle error \*/    } **else** {      usum = ui\_a + ui\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** Architect and Design for Security Policies. Keeping code from wrapping ensure stable structures and allows us to design with security measures in mind. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrèe | 22.04 | Integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 7.3p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | FIO30-C | Exclude user input from format strings |

| **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 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):** Validate Input Data. We need to exclude user input from format strings to avoid breaks or gaps in security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially Implemented |
| CodeSonar | 7.3p0 | IOINJ.FMT  MISC.FMT | Format string injection  Format String |
| Coverity | 2017.07 | TAINTED\_STRING | IMPLEMENTED |
|  |  |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR51-CPP | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** Heed Compiler warnings. Creating a std::string from a null pointer will throw a compiler warning and this should be addressed before pushing code to prod. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Assert\_failure |  |
| CodeSonar | 7.3p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2023.1 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2023.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | FIO39-C. | Do not alternately input and output from a stream without an intervening flush or positioning call |

| **Noncompliant Code** |
| --- |
| This noncompliant code example appends data to a file and then reads from the same file: |
| #include <stdio.h>    **enum** { BUFFERSIZE = 32 };    **extern** **void** initialize\_data(**char** \*data, **size\_t** size);    **void** func(**const** **char** \*file\_name) {  **char** data[BUFFERSIZE];  **char** append\_data[BUFFERSIZE];  **FILE** \*file;      file = **fopen**(file\_name, "a+");  **if** (file == NULL) {      /\* Handle error \*/    }      initialize\_data(append\_data, BUFFERSIZE);    **if** (**fwrite**(append\_data, 1, BUFFERSIZE, file) != BUFFERSIZE) {      /\* Handle error \*/    }  **if** (**fread**(data, 1, BUFFERSIZE, file) < BUFFERSIZE) {      /\* Handle there not being data \*/    }    **if** (**fclose**(file) == EOF) {      /\* Handle error \*/    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, fseek() is called between the output and input, eliminating the undefined behavior: |
| #include <stdio.h>    **enum** { BUFFERSIZE = 32 };  **extern** **void** initialize\_data(**char** \*data, **size\_t** size);    **void** func(**const** **char** \*file\_name) {  **char** data[BUFFERSIZE];  **char** append\_data[BUFFERSIZE];  **FILE** \*file;      file = **fopen**(file\_name, "a+");  **if** (file == NULL) {      /\* Handle error \*/    }      initialize\_data(append\_data, BUFFERSIZE);  **if** (**fwrite**(append\_data, BUFFERSIZE, 1, file) != BUFFERSIZE) {      /\* Handle error \*/    }    **if** (**fseek**(file, 0L, SEEK\_SET) != 0) {      /\* Handle error \*/    }    **if** (**fread**(data, BUFFERSIZE, 1, file) != 0) {      /\* Handle there not being data \*/    }    **if** (**fclose**(file) == EOF) {      /\* Handle error \*/    }  } |

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

| **Principles(s):** Sanitize Data sent to Other Systems. Inputting and outputting data without intervening flush or positioning call can complicate data being sent to other systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported, but no explicit checker |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO39 |  |
| CodeSonar | 7.3p0 | IO.IOWOP  IO.OIWOP | Input After Output Without Positioning  Output After Input Without Positioning |
| Compass/ROSE |  |  | Can detect simple violations of this rule |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM52-CPP | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

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

| **Principles(s):** Use effective quality Assurance. Memory allocation errors can lead to poor performing code so by ensuring we take care of these early we will maintain high QA. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  |  |
| Coverity | 7.5 | CHECKED\_RETURN | Finds Inconsistencies in how function call return values are handled |
| Helix QAC | 2023.1 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| Klocwork | 2023.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | EXP53-CPP. | Do not read uninitialized memory |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    **void** f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>    **void** f() {  **int** i = 0;    std::cout << i;  } |

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

| **Principles(s):** Default Deny. By default we should deny or not read any memory that isn’t initialized as we can’t be sure it is secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Uninitialized-read | Partially checked |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| CodeSonar | 7.3p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| Helix QAC | 2023.1 | DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978 |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP. | Handle all exceptions |

| **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):** Use Effective Quality Assurance. Handling all exceptions ensures the highest quality of code we can output. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2023.1 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | OOP56-CPP. | Honor replacement handler requirements |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a replacement new\_handler is written to attempt to release salvageable resources when the dynamic memory manager runs out of memory. However, this example does not take into account the situation in which all salvageable resources have been recovered and there is still insufficient memory to satisfy the allocation request. Instead of terminating the replacement handler with an exception of type std::bad\_alloc or terminating the execution of the program without returning to the caller, the replacement handler returns as normal. Under low memory conditions, an infinite loop will occur with the default implementation of ::operator new(). Because such conditions are rare in practice, it is likely for this bug to go undiscovered under typical testing scenarios. |
| #include <new>    **void** custom\_new\_handler() {    // Returns number of bytes freed.  **extern** std::**size\_t** reclaim\_resources();    reclaim\_resources();  }    **int** main() {    std::set\_new\_handler(custom\_new\_handler);      // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, custom\_new\_handler() uses the return value from reclaim\_resources(). If it returns 0, then there will be insufficient memory for operator new to succeed. Hence, an exception of type std::bad\_alloc is thrown, meeting the requirements for the replacement handler. |
| #include <new>    **void** custom\_new\_handler() noexcept(**false**) {    // Returns number of bytes freed.  **extern** std::**size\_t** reclaim\_resources();  **if** (0 == reclaim\_resources()) {  **throw** std::bad\_alloc();    }  }    **int** main() {    std::set\_new\_handler(custom\_new\_handler);      // ...  } |

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

| **Principles(s):** Keep it Simple. By honoring replacement handler requirements we make sure our code doesn’t get over complicated by having to perform extra tasks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.1 | DF4776, DF4777, DF4778, DF4779 |  |
| Parasoft  C/C++ test | 2022.2 | CERT\_CPP-OOP56-a  CERT\_CPP-OOP56-b  CERT\_CPP-OOP56-c | Properly define terminate handlers  Properly define unexpected handlers  Properly define new handlers |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | CON55-CPP | Preserve thread safety and liveness when using condition variables |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses five threads that are intended to execute sequentially according to the step level assigned to each thread when it is created (serialized processing). The currentStep variable holds the current step level and is incremented when the respective thread completes. Finally, another thread is signaled so that the next step can be executed. Each thread waits until its step level is ready, and the wait() call is wrapped inside a while loop, in compliance with CON54-CPP. Wrap functions that can spuriously wake up in a loop. |
| #include <condition\_variable>  #include <iostream>  #include <mutex>  #include <thread>    std::mutex mutex;  std::condition\_variable cond;    **void** run\_step(**size\_t** myStep) {  **static** **size\_t** currentStep = 0;    std::unique\_lock<std::mutex> lk(mutex);      std::cout << "Thread " << myStep << " has the lock" << std::endl;    **while** (currentStep != myStep) {      std::cout << "Thread " << myStep << " is sleeping..." << std::endl;      cond.wait(lk);      std::cout << "Thread " << myStep << " woke up" << std::endl;    }      // Do processing...    std::cout << "Thread " << myStep << " is processing..." << std::endl;    currentStep++;      // Signal awaiting task.    cond.notify\_one();      std::cout << "Thread " << myStep << " is exiting..." << std::endl;  }    **int** main() {    constexpr **size\_t** numThreads = 5;    std::**thread** threads[numThreads];      // Create threads.  **for** (**size\_t** i = 0; i < numThreads; ++i) {      threads[i] = std::**thread**(run\_step, i);    }      // Wait for all threads to complete.  **for** (**size\_t** i = numThreads; i != 0; --i) {      threads[i - 1].join();    }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses notify\_all() to signal all waiting threads instead of a single random thread. Only the run\_step() thread code from the noncompliant code example is modified. |
| #include <condition\_variable>  #include <iostream>  #include <mutex>  #include <thread>    std::mutex mutex;  std::condition\_variable cond;    **void** run\_step(**size\_t** myStep) {  **static** **size\_t** currentStep = 0;    std::unique\_lock<std::mutex> lk(mutex);      std::cout << "Thread " << myStep << " has the lock" << std::endl;    **while** (currentStep != myStep) {      std::cout << "Thread " << myStep << " is sleeping..." << std::endl;      cond.wait(lk);      std::cout << "Thread " << myStep << " woke up" << std::endl;    }      // Do processing ...    std::cout << "Thread " << myStep << " is processing..." << std::endl;    currentStep++;      // Signal ALL waiting tasks.    cond.notify\_all();      std::cout << "Thread " << myStep << " is exiting..." << std::endl;  }    // ... main() unchanged ... |

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

| **Principles(s):** Practice Defense in Depth. Thread safety is key and we must be sure we have a level of defense in place to keep threads clean. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | CONCURRENCY.BADFUNC.CNDSIGNAL | Use of Condition Variable Signal |
| Helix QAC | 2023.1 | C++1778, C++1779 |  |
| Klocwork | 2023.1 | CERT.CONC.UNSAFE\_COND\_VAR |  |
| Parasoft  C/C++test | 2022.2 | CERT\_CPP-CON55-a | Do not use the 'notify\_one()' function when multiple threads are waiting on the same condition variable |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | CTR51-CPP | Use valid references, pointers, and iterators to reference elements of a container |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, pos is invalidated after the first call to insert(), and subsequent loop iterations have undefined behavior. |
| #include <deque>    **void** f(**const** **double** \*items, std::**size\_t** count) {    std::deque<**double**> d;    auto pos = d.begin();  **for** (std::**size\_t** i = 0; i < count; ++i, ++pos) {      d.insert(pos, items[i] + 41.0);    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, pos is assigned a valid iterator on each insertion, preventing undefined behavior. |
| #include <deque>    **void** f(**const** **double** \*items, std::**size\_t** count) {    std::deque<**double**> d;    auto pos = d.begin();  **for** (std::**size\_t** i = 0; i < count; ++i, ++pos) {      pos = d.insert(pos, items[i] + 41.0);    }  } |

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

| **Principles(s):** Keep it simple. Using valid pointers, references, and iterators keeps our code simplified, clean and effective. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2023.1 | DF4746, DF4747, DF4748, DF4749 |  |
| Klocwork | 2023.1 | ITER.CONTAINER.MODIFIED |  |

### 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.

\*We should begin implementing automation from the pre-production all the way through the production cycles. Asses and Plan is one step where automation should not take place as we need all hands on deck to plan and prepare for any new threats and where security should be implemented or improved. Verification and testing can be automated through several of the automations tools listed in this policy document above. As it moves from pre-production to production, we should look to automate the transition and health check, as well as the monitoring and detecting stage. However, response, and maintenance should be left to manual processes.

### 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 | P7 | 2 |
| INT30-C | High | Likely | High | P1 | L2 |
| FIO30-C | High | Likely | Medium | P3 | L1 |
| STR51-CPP | High | Likely | Medium | P3 | L1 |
| FIO39-C | Low | Likely | Medium | P8 | L2 |
| MEM52-CPP | High | Likely | Medium | P3 | L1 |
| EXP53-CPP | High | Probable | Medium | P6 | L1 |
| ERR51-CPP | Low | Probable | Medium | P10 | L3 |
| OOP56-CPP | Low | Probable | High | P9 | L3 |
| CON55-CPP | Low | Unlikely | Medium | P11 | L4 |
| CTR51-CPP | High | Probable | High | P2 | L2 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This is encrypting data even when it’s not being used or transferred. Data resting on a hard drive for example should be encrypted in case it were to be stolen then the hacker would have to decrypt the data without a decryption key. |
| Encryption at flight | This is referring to any data moving over a Network or Cloud based organization. Encrypting this is vital as these networks can be hacked and monitored by attackers so any non-encrypted data could easily be stolen. |
| Encryption in use | This is essentially the policy that is ensuring any data no matter what stage it’s in is encrypted so that it cannot be stolen or decrypted at any point by an unauthorized individual. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This refers to having users use valid log in credentials, preferably even 2 factor authentication so that we can validate that only authorized individuals can log in and access our data. |
| Authorization | Authorization refers to maintaining authorized personnel who only have permissions that are accurate to their needs in our systems. |
| Accounting | Reference to needing to continuously audit our users credentials, permissions and authorizations so that security passwords change frequently and that no users are given access that don’t need it or are no longer with the organization. |

### 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 | 3/17/2023 | Milestone 1 | Sam Dayball |  |
| 1.1.1 | 4/4/2023 | Project 1 | Sam Dayball |  |
| 1.1.2 | 7/12/2023 | CS499 Enhancement | Sam Dayball |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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