**DGreen 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 | Validation of input data from all sources helps to eliminate most software vulnerabilities and security breaches. Especially untrusted data sources or external sources, it is critical to monitor and validate incoming data from points of entry like command line arguments, network interfaces, user-controlled files, and environmental variables. |
| 1. Heed Compiler Warnings | Compiler warnings are set in place to help reduce security risks with code. Utilizing static and dynamic analysis tools along with compiler checks and manual review of code to help reduce security flaws in the code. |
| 1. Architect and Design for Security Policies | Creation of a solid foundational software architecture and design of software implementation to incorporate the highest safety level standards. The architecture should reflect proper security at proper levels of authority. Having separate permissions and available resources per tier of authentication is an example. |
| 1. Keep It Simple | As the name implies, the simpler the program or software is, the easier it is to keep safe. It should be as simple as possible while keeping the needs of the client/requirements at peak efficiency. The more complex the program or software is, the harder it is to keep safe and maintain to the highest degree of integrity. |
| 1. Default Deny | The philosophy of access decisions being based on permission instead of exclusion. By default, all members are denied access unless the security scheme provides permissions to the user. |
| 1. Adhere to the Principle of Least Privilege | Every user and process should be given the bare minimum requirements to finish a task. The default is no permissions where on an increasing tier list of authority based on required functions to be performed, little by little permissions are granted based on increasing authority levels. |
| 1. Sanitize Data Sent to Other Systems | All data moving to other systems must be sanitized. Examples include command shells, relational databases, and commercial off the shelf components. Due to potential security risks of unused or improperly utilized security and data on these systems, sanitizing the data beforehand helps mitigate risks with security breaches. |
| 1. Practice Defense in Depth | The practice of utilizing multiple layers of defense with different targeted focuses to create a multi-layered defense that provides protection from all types of threats. It allows for total protection in places where other areas may have shortcomings, the other layers will catch the issues and stop them. |
| 1. Use Effective Quality Assurance Techniques | Proper quality assurance techniques help alleviate the potential security compromised systems by finding the issues before they happen. Tests like fuzz testing, penetration testing, source code audits, and Junit testing all are items that should be embedded within the development cycle and quality assurance protocols. Also, the use of outside perspectives with a negative scope allow the reviewer to come in expecting to find something wrong with no biases, allowing the individual to not overlook anything. |
| 1. Adopt a Secure Coding Standard | Creation and utilization of a secure coding standard allows for efficient and safe creation of programs and software. A universal standard is easy to measure, review, and analyze against current, prior, and/or future production of products to ensure the most safe and efficient product is produced. |

### 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** | **Utilize appropriate data types within their parameters** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Each data type has its own parameters of what values it can take in. Char allows for individual characters within single quotes. String allows for a variety of various characters with almost no bounds as long as it’s within the double quotes. Integers have various types of types that have their own limitations.  Signed and unsigned integers are the first types. Signed is any value, negative or positive (depending on constraints). Unsigned integers start at 0 and are only positive numbers (depending on constraints).  Int is set for 4 bytes worth of data, ranging -2147483648 to 2147483647 for signed and 0 to 4294967295 unsigned. Short is set for 2 bytes, and long is set to 8 bytes.  Not utilizing the appropriate data type can cause overflow if not done properly. SEI CERT C rule INT32-C defines issues with overflow.  INT32-C states that overflow is an undefined behavior within C/C++ language. It states undefined behavior as “Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the C Standard imposes no requirements. An example of undefined behavior is the behavior on integer overflow.” (SEI CERT C Coding Standard, 2023). Overflow causes the intended value to not store all the data from the intended value and will instead wrap around the values outside the limits, which will cause incorrect values resulting in failure of application desired results and promote security weak points. That or it will simply crash the intended function all together.  Ensuring the proper data type is used for each intended use of the value keeps the code operating within its respectable bounds and allows the program to run as intended. |

| **Noncompliant Code** |
| --- |
| The code below has no checks in place to ensure the values are within the parameters of signed int, which may result in overflow of a positive or negative value out of signed integers limits. |
| **\*SEI CERT\***  **void** func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The code below utilizes the header limits.h to provide limit checking tools. The code below provides an if/else clause that checks for values within parameter limits. If the value is less than the integer minimum or greater than the integer maximum, it throws an error. Else, it allows the program to function as intended. |
| **\*SEI CERT\***  #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):** Validate input data applies here as ensuring the proper data type being utilized within the process will allow for the program to perform as expected. Heed compiler warnings will also help ensure that the proper data type is in use as a reminder to the developer to utilize appropriate data types for the designated calls. |
| --- |

**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) | 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](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **TAINTED\_SCALAR**  **BAD\_SHIFT** | Implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **493 S, 494 S** | Partially implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Utilize appropriate data values within their parameters** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Each data type has its own parameters of what values it can take in. Char allows for individual characters within single quotes. String allows for a variety of various characters with almost no bounds as long as it’s within the double quotes. Integers have various types of types that have their own limitations.  Signed and unsigned integers are the first types. Signed is any value, negative or positive (depending on constraints). Unsigned integers start at 0 and are only positive numbers (depending on constraints).  Use of improper data values based on set parameters may cause wrapping within program computations and use of these values, which is a rule standard explained by INT30-C by SEI Cert C ruling.  INT30-C defines wrapping as “Computation involving unsigned operands whose result is reduced module the number that is one greater than the largest values that can be represented by the resulting type.” (SEI CERT C Coding Standard, 2023). This means that when the values hit the limits of the data type, instead of saying out of limits or an error, the value will basically restart at 0 and count back up, or “wrap” around. This can cause values to not produce predicted results and promote failure of programs and security weak points.  Ensuring the proper data value is used for each intended use of the value keeps the code operating within its respectable bounds and allows the program to run as intended. |

| **Noncompliant Code** |
| --- |
| Below the code allows for wrapping as there is no upper bound limit. This can either cause memory to be seized up from other applications or expose a weak point for security breach. |
| **\*SEI CERT\***  **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This code includes the header limits to provide functionality of checking limits of values. The code below has an if/else clause that checks if the value is within limits. If it is not, it throws an error. Else, it allows the program to function as intended. |
| **\*SEI CERT\***  #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):** Validate input data applies here as ensuring the proper data value being utilized within the process will allow for the program to perform as expected. |
| --- |

**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 |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=125337650) | 7.2.0 | **CertC-INT30** | Implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/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](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **INTEGER\_OVERFLOW** | Implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Ensure string usage and storage requirements are met** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Strings are a data type that allow for the input of various characters, integers, and symbols and essentially have no limit on size unless determined by the program and/or system memory. However there are standards with using strings to ensure they are not intentionally or unintentionally abused.  SEI CERT C rules STR30-C, STR31-C/STR50-CPP, and STR32-C are all rules that help with the format and manipulation with strings in C/C++ to ensure there are no errors in operation for best practices in coding and coding security.  STR30-C states “Do not attempt to modify string literals” (SEI CERT C Coding Standard, 2023). When a string literal is created, it’s best not to reference it for editing. This will cause memory pointer failures which can cause the program to malfunction or produce an error. Instead its best to create a new string array and modify the new array. |

| **Noncompliant Code** |
| --- |
| The code below is referencing a string literal and trying to modify it. This produces undefined behavior within the program. |
| **\*SEI CERT\***  **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Below the code creates a string literal that is placed within an array which will produce a copy of the string literal that can be manipulated safely. |
| **\*SEI CERT\***  **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** Validating input data will help determine the correctness of strings as well as heeding compiler warnings will help ensure the code itself is accurate. Practicing defense in depth can help incoming malicious data be mitigated properly as inputs via characters and strings is a common access point. |
| --- |

**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=222953724) | 22.10 | **assert\_failure** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.MEM.NPD** | Null Pointer Dereference |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR51-a** | Avoid null pointer dereferencing |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C++: STR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr51cpp.html) | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Utilize proper structure and security for data entry structures** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL Injection utilizes attack vectors (malicious string/character input vectors/arrays) to gain unauthorized entry via forcing overflow on user entry fields, causing the program to either malfunction or crash, potentially allowing the user to gain access to sensitive information. To protect against this, developers must provide proper buffer protection and null-point termination to strings to stop overflow from happening.  SEI CERT C/CPP rules STR50-CPP/STR31-C and STR32-C are guidelines to prevent overflow and promote proper functionality of strings. STR50-CPP/STR31-C state to guarantee sufficient space within the storage for character data and the null terminator. When the compiler does not see the null pointer, it cannot find the end of the string, and possibly cause overflow. STR32-C states to not pass a non-null-terminated character sequence into a library that’s supposed to be a string. This will cause the same issue.  Ensuring the code has proper storage space and space for the null-termination character ensures all the data is read safely. |

| **Noncompliant Code** |
| --- |
| The code below is one off from having proper storage. This example shows that with the ‘for’ loop it is ranging the full length of n, not providing space for the null-termination character. This may allow for buffer overflow. |
| **\*SEI CERT\***  **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** |
| --- |
| The code below iterates one short of the length of ‘n’ allowing for space of the null-termination character. |
| **\*SEI CERT\***  **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):** Validating input data will help determine the correctness of strings and character arrays as well as heeding compiler warnings will help ensure the code itself is accurate. Practicing defense in depth can help incoming malicious data be mitigated properly as inputs via characters and strings is a common access point in SQL injections. |
| --- |

**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=222953724) | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO LANG.MEM.TO** | No space for null terminator  Buffer overrun Type overrun |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **489 S, 66 X, 70 X, 71 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR50-b** **CERT\_CPP-STR50-c** **CERT\_CPP-STR50-e** **CERT\_CPP-STR50-f** **CERT\_CPP-STR50-g** | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Utilize proper allocation and freeing of memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Appropriate memory allocation and proper freeing of unused memory supports proper memory usage and security. Double-free or writing to already freed memory may result in corruption in the program and unauthorized access within. Dereferencing null pointers or invalid pointers will also cause improper program operation and potential security breach. Buffer overflow is another form of memory exploit that targets char/string values with insufficient space.  SEI CERT C/CPP has a few rules that help prevent these issues: MEM50-CPP Do not access freed memory, EXP53-CPP Do not read uninitialized memory, MEM52-CPP, Detect and handle memory allocation errors, and MEM31-C Free dynamically allocated memory when no longer needed.  Following these forms of memory protection as well as the use of encrypted pointers and overflow prevention measures will help protect against memory attacks. |

| **Noncompliant Code** |
| --- |
| The code below frees the pointer ‘p’ before p->next is executed, causing it to reference freed memory instead of a valid memory address. |
| **\*SEI CERT\***  #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** |
| --- |
| The code below stores a reference p->next into q before freeing p. |
| **\*SEI CERT\***  #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):** Heed compiler warnings and use effective quality assurance techniques will help with proper memory management as these both will help test for proper memory usage, clearing, and allocation/deallocation within the program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Coverity) | 7.5 | **CHECKED\_RETURN** | Finds inconsistencies in how function call return values are handled |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **45 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MEM52-a** **CERT\_CPP-MEM52-b** | Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: MEM52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem52cpp.html) | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Utilize assertions for diagnostic testing of code functionality** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are statements that assert an expression to always be true or must be true. If it is not, it will throw an error and terminate the program. These should only be used for code testing in a manner which they are always true unless there is a bug within the code. It excels in identifying code errors, depreciated errors, or other potential errors from random events that may occur.  This does not replace all diagnostic functionality but should be a tool and generally should be placed after a condition that should’ve been handled properly already. SEI CERT MSC11-C states to incorporate diagnostic testing using assertions. |

| **Noncompliant Code** |
| --- |
| The code below shows the usage of an assertion to verify invalid user input which is not a valid assessment due to assertions not accurately assessing runtime error checking. |
| #include <stdlib.h>  #include <string.h>  Int user\_input;  Cout << “Please enter a value greater than 10” << endl;  Cin >> user\_input;  Assert (user\_input < 10); |

| **Compliant Code** |
| --- |
| The code below checks that the int value ‘a’ is always 10 as it is a constant within this application. |
| Int a = 10;  Int b = 5;  Int c = a + b;  Assert(a == 10); |

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

| **Principles(s):** Heed compiler warnings and use effective quality assurance techniques will help with proper setup of assert statements within the testing portion of the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.FUNCS.ASSERTS** | Not enough assertions |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **ASSERT\_SIDE\_EFFECT** | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-MSC11-a** | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Utilize proper handling techniques for exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions should be thrown in situations where the called function has something preventing it from executing in completeness, meaning if there is some form of error within a function but the whole function can execute there should only be an exception for function critical components.  Exceptions excel within the applications of checking for input validation and other runtime error catching situations. These should be executed within a try/catch setup. Trying a series of code within a function and catch any form of errors and throwing an exception clause is the best method with the application of exceptions within code. When possible, it is preferred to have the exception thrown by value and catch by reference or constant reference to ensure the compiler takes care of memory management efficiently.  SEI Cert CPP/C standards that handle these situations are as follow: ERR61-CPP Catch exceptions by lvalue reference, ERR51-CPP Handle all exceptions, ERR57-CPP Do not leak resources when handling exceptions, and ERR54-CPP Catch handlers should order their parameter types from most derived to least derived.  These rules ensure that the exceptions are all handled properly. If exceptions are not handled properly, they can crash the program and/or caused unexpected behavior within the application. |

| **Noncompliant Code** |
| --- |
| The code below shows a throw exception function that is not properly handled. Without a try/catch method, it will wind up the code from the exception, usually calling out main and terminate the program. |
| #include <iostream>  #include <stdexcept>  Void testValues() {    Bool wrongValues = true;  If(wrongValues) {  Throw std::runtime\_error(“ERROR”);  }  }  Int main() {  testValues();  Std::cout << “This code will not execute if line above is executed” << std::endl;  } |
|  |

| **Compliant Code** |
| --- |
| [Compliant description] |
| #include <iostream>  #include <stdexcept>  Void testValues() {    Bool wrongValues = true;  If(wrongValues) {  Throw std::runtime\_error(“ERROR”);  }  }  Int main() {  Try {  testValues();  }  Catch (const std::runtime\_error& e) {  Std::cerr << “Runtime Error: “ <<e.what() << std::endl;  }  Std::cout << “Program has correct values” << std::endl;  } |

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

| **Principles(s):** Validate input data, practice defense in depth, and use effective quality assurance techniques will help enforce proper use of exceptions. The exceptions can catch and verify input data, and providing enough exception catching will help promote defense in depth as well as the effective use of quality assurance within the code, trying to capture as many possible failures within reason. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | cert-err34-c | Checked by clang-tidy; only identifies use of unsafe C Standard Library functions corresponding to ERR34-C |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **BADFUNC.ATOF BADFUNC.ATOI BADFUNC.ATOL BADFUNC.ATOLL** | Use of atof Use of atoi Use of atol Use of atoll |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR62-a** | The library functions atof, atoi and atol from library stdlib.h shall not be used |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR62-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr62cpp.html) | Checks for unvalidated string-to-number conversion (rule fully covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Utilize proper expression management to deter poor behavior** |
| --- | --- | --- |
| Expressions | [STD-008-CPP] | Expressions need to utilize proper memory management and proper reference to still present values instead of values that have either moved or been removed. Errors with this concept can cause unexpected behavior or prevent the application from running. It may even lead to security breaches as malicious users may be able to exploit the abandoned memory or poor memory from improper expressions and modify them to their purposes.  SEI Cert CPP/C rules that adhere to this idea are as follows: EXP35-C Do not modify objects with temporary lifetime, EXP63-CPP Do not rely on the value of a moved-from object, EXP51-CPP. Do not delete an array through a pointer of the incorrect type, EXP53-CPP Do not read uninitialized memory, EXP54-CPP Do not access an object outside of its lifetime.  These standards ensure that expressions are not modifying values outside of proper lifecycle and memory points are being properly used. |

| **Noncompliant Code** |
| --- |
| The code below is trying to use an uninitialized local variable ‘a’ which will cause undefined behavior. |
| #include <iostream>  Void printa() {    Int a;  Std::cout << a << std::endl;  } |

| **Compliant Code** |
| --- |
| The code below shows a properly instantiated variable ‘a’. |
| #include <iostream>  Void printa() {    Int a = 1;  Std::cout << a << std::endl;  } |

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

| **Principles(s):** Validate input data will help ensure that the expression will take in the proper information and thus perform as expected. Heeding complier warnings will help ensure the proper structure for the expression as well as help with memory management to help keep the memory allocations and accesses functional and appropriate. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | **P8** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.MEM.NPD LANG.MEM.UVAR** | Null Pointer Dereference Uninitialized Variable |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **DF4701, DF4702, DF4703** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-EXP63-a** | Do not rely on the value of a moved-from object |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: EXP63-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp63cpp.html) | Checks for read operations that reads the value of a moved-from object (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Utilize proper container handling methods to prevent unexpected behavior** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Containers are the most efficient method of storing values with proper memory management. Depending on the desired application of these values will determine which type of container to use, each having their own syntax and parameters. If the parameters utilize improper data types or references, it can cause unexpected behavior.  The wrong container used for various applications will cause issues with calling the data later. Using a map instead of a char array for characters may cause referencing issues depending on the application and unexpected behavior.  It is important to check that container indices and iterators are within the valid range of the container. Anything that may go out of bounds will cause unexpected behavior.  SEI Cert CPP/C standards relevant to this topic include: CTR50-CPP Guarantee that container indices and iterators are within the valid range, CTR51-CPP Use valid references, pointers, and iterators to reference elements of a container, CTR52-CPP Guarantee that library functions do not overflow, CTR53-CPP Use valid iterator ranges. |

| **Noncompliant Code** |
| --- |
| The code below shows a function that adds a value into the table. The pos location of the table is a signed int instead of unsigned or a declaration that only allows positive values. This will cause unexpected behavior if pos goes below 0 as there is no catch for a non-appropriate table value. |
| \*SEI CERT\*  #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, **int** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| The code below changes the pos to a size\_t ensuring it is always a positive value. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, std::**size\_t** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

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

| **Principles(s):** Validating input data will help ensure the proper limits and types of data are used for the containers to promote proper functionality. Heeding compiler warnings about the container structure as well as using appropriate quality assurance techniques and defense in depth will help provide a secure platform to help stop malicious intent, and in the event of an attack, handle the attack securely by having multiple avenues to stop the abuse of the container. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.MEM.BO LANG.MEM.BU LANG.MEM.TO LANG.MEM.TU LANG.MEM.TBA LANG.STRUCT.PBB LANG.STRUCT.PPE LANG.STRUCT.PARITH** | Buffer overrun Buffer underrun Type overrun Type underrun Tainted buffer access Pointer before beginning of object Pointer past end of object Pointer Arithmetic |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-CTR50-a** | Guarantee that container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: CTR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr50cpp.html) | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Utilize the proper methodology and practices of Object Orientated Programming.** |
| --- | --- | --- |
| Object Orientated Programming | [STD-010-CPP] | Object Orientated Programming (OOP) is not a guaranteed safe method of coding, but more to help keep the code organized efficiently and effectively. How an individual may utilize the methodology within their manner of coding will determine its effectiveness and safety. Ensuring all methods are used properly will ensure that the OOP works as intended.  SEI Cert CPP standards that are relevant include: OOP50-CPP. Do not invoke virtual functions from constructors or destructors,  OOP51-CPP. Do not slice derived objects, OOP52-CPP. Do not delete a polymorphic object without a virtual destructor, OOP53-CPP. Write constructor member initializers in the canonical order, OOP54-CPP. Gracefully handle self-copy assignment, OOP55-CPP. Do not use pointer-to-member operators to access nonexistent members, OOP56-CPP. Honor replacement handler requirements, OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions, OOP58-CPP. Copy operations must not mutate the source object. |

| **Noncompliant Code** |
| --- |
| The code below initializes a value for the list C::C() but initializes values in incorrect order. The second value is being initialized first which will have an unspecified value going to the second initialized value which should be the first. |
| **class** C {  **int** dependsOnSomeVal;  **int** someVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| The code below has the values initialized in the proper order within the list C::C(). |
| **class** C {  **int** someVal;  **int** dependsOnSomeVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** Architect and design for security practices and keep it simple will work together in not only ensuring that the code itself is structured properly, but to keep it as precise and efficient as possible to mitigate risks with either non-conforming code and/or poor code quality. Defense in depth can be utilized within this standard as well as there can be multiple variations of providing defense in development as well as the finished product, covering many facets of potential security issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **non-virtual-public-destructor-in-non-final-class** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.STRUCT.DNVD** | delete with Non-Virtual Destructor |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-OOP52-a** | Define a virtual destructor in classes used as base classes which have virtual functions |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: OOP52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcoop52cpp.html) | Checks for situations when a class has virtual functions but not a virtual destructor (rule partially covered) |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation can be enforced effectively within the build and monitor and detect sections of the diagram above. Within the build portion, many manual processes that are either simple and/or redundant could be automated to help reduce development time. For monitoring and detection, several automated processes and tests could be implemented to identify common issues and provide simple diagnostic capabilities to help with troubleshooting and error management.

### 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 | Likely | High | **P9** | **L2** |
| STD-002-CPP | High | Likely | High | **P9** | **L2** |
| STD-003-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-004-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-005-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-006-CPP | Low | Unlikely | High | **P1** | **L3** |
| STD-007-CPP | Medium | Unlikely | Medium | **P4** | **L3** |
| STD-008-CPP | Medium | Probable | Medium | **P8** | **L2** |
| STD-009-CPP | High | Likely | High | **P9** | **L2** |
| STD-010-CPP | Low | Likely | Low | **P9** | **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 at rest | Data at rest will be encrypted utilizing full-disk encryption at the server level and at the database level will utilize server-wide encryption policies. Both will utilize further backup strategies for case of major failure and recovery. |
| Encryption in flight | Data in flight will be encrypted utilizing public key infrastructure for incoming/outgoing data exchanges including but limited to messages, attachments, file transfers, web traffic, and all forms of communication. SSH, password access, and restrictive communication channels will be in use to prevent malicious activity. |
| Encryption in use | Data in use will utilize identity security protocol to ensure the correct person/body is allowed access only when all fields of identification are satisfied, and least privileged role authority to only allow access for only necessary actions and functions. Utilizing a system with multi-tiered approval for access change requests will help ensure levels of access are cross-examined to keep sensitive data secure. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of identifying a user, usually done via username/password. This controls whether the user has access to the system or not to ensure only authorized personnel are given access to the system. |
| Authorization | Authorization provides access on a least-privilege system to authenticated users to control the level of mobility and authority they have over the system. This also allows for the addition of new users, modification of access, or full removal of access. |
| Accounting | Accounting is the tracking of user interactions with the system. It monitors the files accessed by users as well as any changes to the database via user interaction. |

**\***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 | 07/21/2024 | First edits within template | James Nikolaou |  |
| 1.2 | 08/14/2024 | Final edits within template | James Nikolaou |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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