**Green Pace Developer: Dylan Ngu**



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc201011636)

[Contents 1](#_Toc201011637)

[Overview 2](#_Toc201011638)

[Purpose 2](#_Toc201011639)

[Scope 2](#_Toc201011640)

[Ten Core Security Principles 2](#_Toc201011641)

[C/C++ Ten Coding Standards 3](#_Toc201011642)

[Coding Standard 1 4](#_Toc201011643)

[Coding Standard 2 6](#_Toc201011644)

[Coding Standard 3 8](#_Toc201011645)

[Coding Standard 4 10](#_Toc201011646)

[Coding Standard 5 12](#_Toc201011647)

[Coding Standard 6 14](#_Toc201011648)

[Coding Standard 7 16](#_Toc201011649)

[Coding Standard 8 18](#_Toc201011650)

[Coding Standard 9 20](#_Toc201011651)

[Coding Standard 10 22](#_Toc201011652)

[Defense-in-Depth Illustration 24](#_Toc201011653)

[Automation 24](#_Toc201011654)

[Summary of Risk Assessments 25](#_Toc201011655)

[Policies for Encryption and Triple A 25](#_Toc201011656)

[Audit Controls and Management 27](#_Toc201011657)

[Enforcement 27](#_Toc201011658)

[Exceptions Process 27](#_Toc201011659)

[Distribution 28](#_Toc201011660)

[Policy Change Control 28](#_Toc201011661)

[Policy Version History 28](#_Toc201011662)

[Appendix A Lookups 28](#_Toc201011663)

[Approved C/C++ Language Acronyms 28](#_Toc201011664)

## 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.

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All input from external sources should be treated as untrusted data and be validated before being used. |
| 1. Heed Compiler Warnings | Use the highest warning level when compiling code. Warnings do not prevent code from compiling, but they indicate problems that could lead errors or unexpected behavior and therefore, must be addressed by modifying source code. |
| 1. Architect and Design for Security Policies | Establish a framework for implementing and enforcing security policies when designing and developing software. This ensures the security policy aligns with business goals and relevant regulations. |
| 1. Keep It Simple | Keep code simple and avoid using complex designs. Complexity can increase the chances of errors and requires more effort to ensure appropriate levels of security. |
| 1. Default Deny | By default, access should be denied until conditions are met. Essentially, access can’t be granted until users are properly authenticated, and their authorization is determined. |
| 1. Adhere to the Principle of Least Privilege | All processes should be executed with the minimal necessary permissions to perform the intended task. Limiting access and permission will reduce the chance for an attacker to compromise a system with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex subsystems must be sanitized to prevent exposing sensitive system information. External systems are unaware of the context in which data is being passed to them, therefore our system is responsible for validating and formatting data being sent. |
| 1. Practice Defense in Depth | Leverage multiple, redundant layers of defense to protect against security risks. This ensures that when one layer is compromised, another layer of defense can prevent a security flaw from being exploited and reduce the potential harm. |
| 1. Use Effective Quality Assurance Techniques | Security shouldn’t be assumed as adequate and quality assurance techniques should be used to identify and eliminate vulnerabilities. Extensive security testing through code audits and external review can lead to more secure systems by analyzing software from different perspectives. |
| 1. Adopt a Secure Coding Standard | Determine an appropriate secure coding standard and apply it during development to ensure properly secured code. Adoption of a secure coding standard provides consistency and security and should be followed by all developers. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Call functions with the correct number and type of arguments.** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not call a function with the wrong number or type of arguments. Invoking a function with an incompatible declaration can result in undefined behavior. This may generate compiler warnings but does not prevent code compilation. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, function f() is defined to take an argument of type long but is called in main() with an argument of type int. This could lead to undefined behavior in more complex scenarios. |
| #include <iostream>  long f(long x){  return x < 0 ? -x : x;  }  int main() {  int myInt = -5;  long result = f(myInt); // type mismatch, implicit conversion  std::cout << “Result: “ << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Compliance could be achieved by modifying f() to accept an int or changing the function call to pass a long to f(). In the compliant code example, function f() is called using an explicitly casted long argument. |
| #include <iostream>  long f(long x){  return x < 0 ? -x : x;  }  int main() {  int myInt = -5;  long result = f(static\_cast<long>(myInt); // explicit conversion  std::cout << “Result: “ << result << std::endl;  return 0;  } |

| **Principles(s):**   1. **Validate Input** – Validating inputs ensure functions are called with the intended number and type of arguments. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | incompatible-argument-type  parameter-match  parameter-match-computed  parameter-match-type | Fully checks argument types and parameter matches |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.11.0 | premium-cert-exp37-c |  |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151949) | 3.11 | [**S930**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-930) | Detects incorrect argument count |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C: Rule EXP37-C](https://www.mathworks.com/help/bugfinder/ref/certcruleexp37c.html) | Checks for standard function calls with incorrect arguments, including implicit function declarations and unreliable cast of function pointers. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Guarantee that container indices and iterators are within the valid range** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Programmers are responsible for ensuring that array references and integer indexes are within bounds. Invalid array or container index can result in arbitrary memory overwrite. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #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** |
| --- |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments |
| #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;  } |

| **Principles(s):**   1. **Validate Input** – Container indices must be validated to ensure they are within the correct range, especially if they are influenced by untrusted sources. |
| --- |

**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=222953724) | 22.10 | overflow\_upon\_dereference |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | CERT\_CPP\_CTR50-a | Guarantees container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [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, and pointer dereference with tainted offset. Rule partially covered |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator.** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, the input is unbounded, which could lead to buffer overflow. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):**   1. **Validate Input** – ensure inputs are of correct size to be stored. 2. **Architect and design for security policies** – Utilizing std::string or other dynamic memory allocation functions can help enforce a security policy |
| --- |

**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 and soundly supported |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | 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' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: STR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr50cpp.html) | Checks for:  Use of dangerous standard function.  Missing null in string array.  Buffer overflow from incorrect string format specifier.  Destination buffer overflow in string manipulation.  Insufficient destination buffer size.  Rule partially covered. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S3519**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-3519) |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude user input from format strings or functions** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not use user-provided input directly within format strings or functions. Doing so could allow an attacker to manipulate format specifiers or inject executable code and compromise a system. Always use string literals for format strings and pass user data as arguments to enforce a clear separation between data elements and executable code. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the unvalidated “username” parameter is simply appended to the query. An attacker can enter SQL code into the query and the application will execute the attackers code on the database. For example, if an attacker enters “‘ OR ‘1’=’1” for “username”, they will bypass authentication because ‘1’=’1’ is always true. |
| *// Get user input for username (vulnerable)*  std::string username;  std::cout << "Enter username: ";  std::getline(std::cin, username);   *// Construct the SQL query using string concatenation*  std::string sql = "SELECT \* FROM users WHERE username = '" + username + "';"; |
| The resulting query sent to the database would be:  SELECT \* FROM users WHERE username = '' OR '1'='1'; |

| **Compliant Code** |
| --- |
| In the compliant code example, we use a parameterized query to separate SQL code from user-supplied input. Instead of directly embedding user input into the SQL query, we create a query template with a place holder (“?”). This method ensures user input is treated only as data. If the attacker were to use the same exploit as in the noncompliant code, the database would search for a user whose username directly matches '' OR '1'='1'. |
| *// Get user input for username (vulnerable)*  std::string username;  std::cout << "Enter username: ";  std::getline(std::cin, username);   *// Construct the SQL query using string concatenation*  std::string sql = "SELECT \* FROM users WHERE username = ?;";    // Assuming a connection object 'conn' is established  sql::PreparedStatement\* stmt = conn->prepareStatement(sql);  stmt->setString(1, username); |

| **Principles(s):**   1. **Validate Input** – verify user supplied inputs are of expected format   3) **Architect and design for security policies** – design functions to use string literals instead of user-supplied input to enforce security policies  7) **Sanitize data sent to other systems** – user supplied inputs must be sanitized to avoid SQL injection |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.11.0 | premium-cert-fio30-c |  |
| [sqlmap](https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://sqlmap.org/&ved=2ahUKEwi31YS18fGNAxUwkokEHRhlHfkQFnoECB0QAQ&usg=AOvVaw1MuunX96N_gnOoSVQRXEm5) | 1.9 |  | Automated testing tool for detecting and exploiting SQL injection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C: Rule FIO30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulefio30c.html) | Checks for tainted string format (rule partially covered) |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Failure to detect memory allocation failures can cause a program to abnormally terminate and denial-of-service-attacks. The default memory allocation operator new() can either throw a bad\_alloc exception or return a nullptr depending on which form is used. Memory allocation failures must be handled appropriately. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an int array is created using new[](std::size) and the results of the allocation are not checked. The function is also marked as noexcept, but new[] can throw an exception if allocation fails, resulting in abnormal termination of the program. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, we use std::nothrow so the new operator will return a null pointer when memory allocation fails. We must then test the returned pointer to ensure it is not null before referencing the pointer. |
| #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;  } |

| **Principles(s):**   1. **Heed compiler warnings** – issues related to memory allocation must be eliminated by modifying code 2. **Use effective quality assurance techniques** – quality assurance techniques can detect memory allocation errors before they can become vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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** | **Do not use assertions for handling runtime errors or input validation** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions should only be used during development to verify conditions that are assumed to be true, not to check for conditions that might occur due to external factors. Assertions are typically disabled in release builds therefore security checks based on assertions will not be active in production, resulting in potential vulnerabilities to be exposed. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, assert() is used to validate the user’s input for age. If this code were in production, assertions would be disabled, resulting in the loss of input validation |
| #include <iostream>  #include <cassert>  int main() {  int age;  std::cout << "Enter your age: ";  std::cin >> age;  // Input validation using assertion  assert(age >= 0 && age <= 120);  std::cout << "Your age is: " << age << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, the user’s input is validated using a conditional statement. If age is not within the acceptable range, the program will handle the error appropriately. |
| #include <iostream>  int main() {  int age;  std::cout << "Enter your age: ";  std::cin >> age;  // Input validation  If (age < 0 || age > 120) {  // handle error  return;  }  std::cout << "Your age is: " << age << std::endl;  return 0;  } |

| **Principles(s):**   1. **Architect and design for security policies** – software must be appropriately designed to enforce security policies including input validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [clang-tidy](https://clang.llvm.org/extra/clang-tidy/) | 21.0.0git | bugprone-assert-side-effect | Finds ‘assert()’ with side effect |
| [cppcheck](https://cppcheck.sourceforge.io/) | 2.17 | CheckAssert | Checking for side effects in assert statements |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions thrown by an application must be caught by a matching exception handler. Improper exception handling will result in abnormal process termination which is a typical vector of denial-of-service attacks. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In the compliant code example, main() handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

| **Principles(s):**   1. **Heed compiler warnings** – compiler warnings about potential exceptions must be handled 2. **Architect and design for security policies** – design software to account for different types of exceptions |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| [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. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Use valid iterator ranges** |
| --- | --- | --- |
| Data Value | [STD-008-CPP] | When iterating over elements of a container, the iterators used must iterate over a valid range. Using a range of two iterators that are invalidated or do not refer into the same container results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, the first iterator representing the start, c.begin(), does not precede the iterator representing the end of the range, c.end(). Each iteration of the loop, for\_each() compares the first iterator (after incrementing it) with the second for equality. Because they will not become equal, the first iterator will continue to be incremented, resulting in undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):**   1. Validate input – valid iterator ranges must be used 2. Keep it simple – leverage standard functions to ensure valid iterator ranges |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | overflow\_upon\_dereference |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | LANG.MEM.BO | Buffer Overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C++: CTR53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr53cpp.html) | Checks for invalid iterator range (rule partially covered). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| String Correctness | [STD-009-CPP] | The std::basic\_string class is paired with std::char\_traits to create the std::string class and the std::char\_traits::length() function is frequently used to determine the number of characters in a null-terminated string. Passing a null pointer into the length() function is undefined behavior because it dereferences a null pointer. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, a std::string object is created from the results of getenv(). This code can lead to undefined behavior because getenv() returns a null pointer on failure. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):**   1. Validate input – ensure external inputs aren’t null before constructing std::string objects |
| --- |

**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/cplusplus/CodeSonar) | 9.0o0 | 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). |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Allocate sufficient memory for an object** |
| --- | --- | --- |
| Memory Protection | [STD-010-CPP] | The allocated memory size must be large enough to hold the object being stored to prevent potential buffer overflow and the execution of arbitrary code. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, the size argument of malloc() is determined by multiplying the user input by sizeof(int). This results in a potential for integer overflow if size is too large |
| int main() {  int size;  std::cout << "Enter the number of elements: ";  std::cin >> size;  int\* arr = (int\*)malloc(size \* sizeof(int));  // ...  free(arr);  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant solution, the user input is checked for potential integer overflow before memory allocation. If size is too large, proper error handling will occur. We can also use the new and delete operators to simplify memory allocation and reduce errors. |
| int main() {  int size;  std::cout << "Enter the number of elements: ";  std::cin >> size;  if (size > std::numeric\_limits<size\_t>::max() / sizeof(int)) {  // handle error  }  int\* arr = new int[size];  // ...  delete[] arr;  return 0;  } |

| **Principles(s):**   1. Validate input – data from external sources must be checked to ensure sufficient memory allocation   3) Architect and design for security policies – ensure software is designed to prevent memory allocation errors |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | malloc-size-insufficient | Partially checked  Besides direct rule violations, all undefined behavior resulting from invalid memory accesses is reported by Astrée. |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.11.0 | premium-cert-mem35-c |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | CERT\_C-MEM35-a | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via 'malloc', 'calloc' or 'realloc' function |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C: Rule MEM35-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemem35c.html) | Checks for:  Pointer access out of bounds  Memory allocation with tainted size  Rule partially covered. |

### Defense-in-Depth Illustration

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



### Automation

Provide a written explanation using the image provided.



Implementing a DevSecOps approach is an effective way to build secure applications, and automation plays a key role in supporting this effort. The security principles and standards outlined in this security policy document should be incorporated throughout the DevSecOps lifecycle. Automation is particularly useful for the build and verify and testing phases during pre-production. By integrating automation into CI/CD pipelines, developers can automate security controls and tests, which promotes consistency and can limit the introduction of vulnerable code earlier in the software development lifecycle. Automation can be used to execute a wide range of tests—such as unit tests, integration tests, UI tests, and end-to-end tests—designed for the application. Beyond functional testing, these tools can also perform static code analysis to identify programming errors or potential vulnerabilities, and they can scan project dependencies for known security risks. Dynamic application security testing (DAST) is also a powerful automated tool that can find potential vulnerabilities when the application is running.

After pre-production, the transition and health check stage can also benefit from automation. The CI/CD pipeline can be used to automate the secure configuration and deployment of the application. After deployment, automated penetration testing tools can assess the system for exploitable weaknesses. Additional automation tools can collect and analyze log data from the application to detect anomalies. If an issue is discovered, alerts can be triggered to inform relevant personnel and initiate predefined automated responses. These responses may include limiting access or isolating vulnerable components in response to a suspected intrusion, aligning with incident response practices. After responding to attacks, embedded automated tools will verify the security of the updated system and continue to monitor for potential exploits. Embedding automated security tools throughout the DevOps process will ensure a more secure system that conforms to standards outlined in this policy.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Probable | Medium | P4 | L3 |
| 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | High | Likely | Medium | P18 | L1 |
| STD-010-CPP | High | Probable | Medium | P6 | L2 |

### Policies for Encryption and Triple A

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 | Encryption at rest refers to encrypting stored data in order to protect it from unauthorized access. This ensures that data remains protected, even if an attacker gains access to data stores, because reading the data can only be done with the decryption key. This policy conforms with our security mission of defense-in-depth because it offers an additional layer of security to protect sensitive information if other security measures were to fail. |
| Encryption in flight | Encryption in flight refers to encrypting data that is being transmitted over a network. This is crucial during transmission to ensure data is unreadable to unauthorized parties who may have intercepted it. Data should be encrypted before leaving its source, using a specific encryption key, then decrypted using a corresponding key when received by the recipient system. This adds another layer of security to ensure data privacy. |
| Encryption in use | Encryption in use refers to the practice of keeping data encrypted while it is being actively used or processed. Encrypting data in use compliments the other two forms and keeps data secure throughout is lifetime. This added layer of security ensures data remains protected while being worked on, even if the system is compromised. This policy also conforms with the sixth secure coding principle because it adheres to the principle of least privilege; data would be used with the least set of privileges necessary to complete a task. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process identifying a user and confirming the user is whom they claim to be. In this process, users request to gain access to the system by using their user login credentials which are then compared to other user credentials that are stored in a database. The user is granted access if their credentials match or denied access if there is a variance and authentication fails. This policy ensures access is denied by default and only granted if security conditions are met. New users can only gain access if administration adds them to the system. |
| Authorization | Authorization is the Triple-A framework strategy of determining what privileges and permissions an authenticated user has. Users of the system are assigned different levels of access, which determine the actions they are allowed to perform as well as the system resources available to them. This policy enforces restrictions and ensures users are only granted access based on their specified roles. Authorization helps to limit the potential damage an attacker could do if they gained access to a compromised account. |
| Accounting | Accounting is the process of providing records of who did what within the system. Once a user is authenticated, all of their activity can be logged in association with that user’s authorization credentials. This policy ensures administrators can view which actions a user performs and when they completed it. Administrators will be able to see when database changes are made, which files have been accessed, and other usage information within the system. Proper accounting can help detect malicious users or activity and provide an audit trail to track all of their actions. This is beneficial for determining which parts of a system have been compromised due to an attack or data breach and helps guide an incident response plan. |

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 | 5/31/2025 | Completed 10 Security Principles & 10 Coding Standards | Dylan Ngu | [Insert text.] |
| 1.2 | 6/17/2025 | Completed Risk Assesment and Automated Detection for each coding standard, Automation, and policies for Encryption and the three elements of the Triple-A Framework | Dylan Ngu | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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