**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 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 | Perform input validation to verify user-entered data is valid before being used by the program. Improper input validation could result in vulnerabilities like SQL injection or cross-site scripting (XSS). Improper input validation could also result in errors caused by unexpected or invalid data. |
| 1. Heed Compiler Warnings | Compiler warnings exist to alert you of a potentially serious problem with your code. While compiler warnings do not stop your code from compiling, ignoring them could result in major errors or security flaws. |
| 1. Architect and Design for Security Policies | By architecting and designing security policies, you are prioritizing security instead of treating it as an afterthought. Refactoring later to account for security practices you should have accounted for is a bad use of development time. Incorporating security policy planning into the design process ensures that you are being proactive about security. |
| 1. Keep It Simple | Simple code is easier to develop, maintain, and troubleshoot than complex code. By keeping the code simple, it is easier to understand how the software functions and ensure your approach is security conscious. |
| 1. Default Deny | By default, access and privileges should be denied. Access should only be granted when necessary. Denial by default helps prevent unintentional holes that could result in unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | By default, only the lowest level of privilege necessary should be granted. Users and applications should not be granted access to systems they do not need to access. This reduces possible attack vectors and limits the damage if something were to be compromised. |
| 1. Sanitize Data Sent to Other Systems | When sending data to other systems, data should always be sanitized to ensure that the data is properly formatted and safe. Without proper sanitization, you leave yourself open to attack vectors like SQL Injection and cross-site scripting (XSS). |
| 1. Practice Defense in Depth | Defense in depth is a strategy where multiple layers of security are used. Each of these layers complement each other to provide additional security. A proper defense in depth implementation makes it much more difficult for attackers as they must bypass multiple security layers. |
| 1. Use Effective Quality Assurance Techniques | Without using proper quality assurance techniques, you cannot be sure that the security measures you’ve implemented are effective. Quality assurance helps you identify defects in the code early in the development process. For this reason, quality assurance should not be an afterthought, but something utilized throughout the development process. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard is important in software development. Secure coding standards help ensure that our code is secure and prevent data breaches, unauthorized access, and other vulnerabilities. Our software being vulnerable has real world implications including loss of privacy, financial loss, and sometimes even loss of life. By adopting a secure coding standard, we prioritize writing secure code and securing our software. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [INT-001-CPP] | When conducting operations involving unsigned integers, ensure they do not wrap by testing if the operation would cause the unsigned integers to wrap and introducing error handling for that scenario. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a wrap on unsigned integer usum will occur when adding ui\_a + ui\_b. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum = ui\_a + ui\_b;  /\* … \*/  } |

| **Compliant Code** |
| --- |
| In this compliant code example, we test whether a wrap would occur in the operation involving unsigned integers before executing the operations. If a wrap would occur, we have included error handling for that scenario. |
| 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):** Use Effective Quality Assurance Techniques – To adhere to this principle, quality assurance techniques should be utilized to detect and prevent integer wraparound from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | cast-integer-to-enum | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 |  |
| CodeSonar | 9.1p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2025.1 | C++3013 |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PVS-Studio | 7.37 | V1016 |  |
| RuleChecker | 22.10 | cast-integer-to-enum | Partially checked |
| Polyspace Bug Finder | R2024b | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [INT-002-CPP] | Avoid casting to an out-of-range enumeration value by performing checks before conversion. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, values are cast to an enumeration type before checking whether the given value is within the range of acceptable values. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  if (enumVar < First || enumVar > Third) {  // Handle Error  }  } |

| **Compliant Code** |
| --- |
| In this compliant code example, we perform a check whether the given value is within the range of acceptable values before converting it to an enumeration type. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle Error  }  EnumType enumVar = static\_cast<EnumType>(intVar); |

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

| **Principles(s):** Validate Input Data – This coding standard can be prevented by performing input validation to ensure a value is within the acceptable range before conversion. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| CodeSonar | 9.1p0 | **MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO** | No space for null terminator Buffer overrun Type overrun |
| Helix QAC | 2025.1 | **C++5216**  **DF2835, DF2836, DF2839,** |  |
| Klocwork | 2025.1 | **NNTS.MIGHT NNTS.TAINTED NNTS.MUST SV.UNBOUND\_STRING\_INPUT.CIN** |  |
| LDRA tool suite | 9.7.1 | **489 S, 66 X, 70 X, 71 X** | Partially implemented |
| Parasoft C/C++test | 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 | R2024b | CERT C++: STR50-CPP | 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. |
| RuleChecker | 22.10 | **stream-input-char-array** | Partially checked |
| Security Reviewer - Static Reviewer | 6.02 | **RTOS\_33 RTOS\_34 shadowVariable UNSAFE\_03 UNSAFE\_04** | Fully implemented |
| SonarQube C/C++ Plugin | 4.10 | **S3519** |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STR-001-CPP] | When using strings, guarantee that there is sufficient storage for both the string and the null terminator. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a buffer overflow could occur if the user input is larger than the buffer. |
| #include <iostream>  void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| In this compliant code example, std::string is used instead of char. |
| #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):** Validate Input Data – this coding standard requires the validation that there is sufficient space for the string being used.  Use Effective Quality Assurance Techniques – Quality assurance techniques should be utilized to detect defects relating to this coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| CodeSonar | 9.1p0 | **MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO** | No space for null terminator Buffer overrun Type overrun |
| Helix QAC | 2025.1 | **C++5216**  **DF2835, DF2836, DF2839,** |  |
| Klocwork | 2025.1 | **NNTS.MIGHT NNTS.TAINTED NNTS.MUST SV.UNBOUND\_STRING\_INPUT.CIN** |  |
| LDRA tool suite | 9.7.1 | **489 S, 66 X, 70 X, 71 X** | Partially implemented |
| Parasoft C/C++test | 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 | R2024b | CERT C++: STR50-CPP | 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. |
| RuleChecker | 22.10 | **stream-input-char-array** | Partially checked |
| Security Reviewer - Static Reviewer | 6.02 | **RTOS\_33 RTOS\_34 shadowVariable UNSAFE\_03 UNSAFE\_04** | Fully implemented |
| SonarQube C/C++ Plugin | 4.10 | **S3519** |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STR-002-CPP] | When passing data to complex subsystems, ensure that data is properly sanitized. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, there is no sanitization being performed on the user input. This could result in SQL Injection through user input. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  void find\_user(sqlite3\* userdb) {  std::string username;  std::cout << “Enter username: “;  std::getline(std::cin, username);  std::string query = “SELECT \* FROM users WHERE username = ‘” + username + “’;  } |

| **Compliant Code** |
| --- |
| In this compliant code example, we use prepared statements instead of directly creating sql queries using user input. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  void find\_user(sqlite3\* userdb) {  std::string username;  std::cout << “Enter username: “;  std::getline(std::cin, username);  std::string query = “SELECT \* FROM users WHERE username = ?;”;  } |

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

| **Principles(s):** Sanitize Data Sent To Other Systems – This coding standard deals with performing sanitization (or avoiding using user input in SQL queries altogether).  Defense in Depth – Even assuming correct input, it is good practice to have layered defense to reduce risk if something should pass your controls. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 9.1p0 | **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 |
| Klocwork | 2025.2 | **NNTS.TAINTED SV.TAINTED.INJECTION** |  |
| LDRA tool suite | 9.7.1 | **108 D, 109 D** | Partially 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 | R2024b | CERT C: Rec. STR02-C | 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** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-001-CPP] | Add code to detect and handle errors relating to memory allocation. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, there is no error handling for errors relating to memory allocation and could result in abrupt crashes. |
| 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 this compliant code example, code was introduced to detect errors relating to memory allocation and handle them properly. |
| #include <cstring>  #include <new>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // Error Handling Code  return  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // …  delete [] copy;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Quality Assurance Techniques can detect and handle errors relating to memory allocation and prevent unexpected behavior, crashes, or vulnerabilities.  Adopt A Secure Coding Standard – When developing with secure coding standards in mind, memory allocation is something that should be accounted for. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  |  |
| Coverity | 7.5 | **CHECKED\_RETURN** | Finds inconsistencies in how function call return values are handled |
| Helix QAC | 2025.2 | **C++3225, C++3226, C++3227, C++3228, C++3229, C++4632** |  |
| Klocwork | 2025.2 | **NPD.CHECK.CALL.MIGHT NPD.CHECK.CALL.MUST NPD.CHECK.MIGHT NPD.CHECK.MUST NPD.CONST.CALL NPD.CONST.DEREF NPD.FUNC.CALL.MIGHT NPD.FUNC.CALL.MUST NPD.FUNC.MIGHT NPD.FUNC.MUST NPD.GEN.CALL.MIGHT NPD.GEN.CALL.MUST NPD.GEN.MIGHT NPD.GEN.MUST RNPD.CALL RNPD.DEREF** |  |
| LDRA tool suite | 9.7.1 | **45 D** | Partially implemented |
| Parasoft C/C++test | 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 |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| PVS-Studio | 7.37 | **V522, V668** |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [ERR-001-CPP] | Prevent abrupt termination by avoiding assertions which call the abort function. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, assertions are used which could result in the program being terminated abruptly. |
| #include <stdio.h>  #include <assert.h>  void print\_number(int\* myInt) {  assert (myInt!=NULL);  printf (\*myInt);  }  int main () {  int \* a = NULL;  print\_number(a);  return 0  } |

| **Compliant Code** |
| --- |
| In this compliant code example, assertions are not used and the integer pointed to NULL is correctly handled. |
| #include <stdio.h>  void print\_number(int\* myInt) {  if (myInt == NULL) {  fprintf(stderr, “Error: NULL pointer passed to print\_number().\n”);  return;  }  Printf(\*myInt);  }  int main() {  int\* a = NULL;  print\_number(a);  return 0;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Effective quality assurance techniques would not rely assertions which cause hard crashes. Instead, testing would utilize testing in a way which fails safely.  Keep It Simple – Taking a more structured approach to error handling makes testing easier to understand.  Adopt A Secure Coding Standard – Secure coding standards would advise against error handling which results in hard crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | **bad-function bad-macro-use** | Supported |
| Compass/ROSE |  |  | Can detect some violations of this rule. However, it can only detect violations involving abort() because assert() is implemented as a macro |
| LDRA tool suite | 9.7.1 | **44 S** | Enhanced enforcement |
| Parasoft C/C++test | 2024.2 | CERT\_C-ERR06-a | Do not use assertions |
| PC-lint Plus | 1.4 | **586** | Fully supported |
| PVS-Studio | 7.37 | **V2021** |  |
| RuleChecker | 24.04 | **bad-function bad-macro-use** | Supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [ERR-002-CPP] | Implement error handling for all exceptions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither of the functions in the example catch exceptions thrown by the throwing\_func() function. This will cause the program to abruptly end. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant code example, a try catch block is added to catch the exception thrown by throwing\_func() and handle the error. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  try {  f();  } catch (…) {  // Handle Error  }  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Quality assurance techniques can be used to verify exceptions are handled safely and identify unhandled exceptions that could lead to crashes, vulnerabilities, or unexpected behavior.  Keep It Simple – By handling all exceptions, we can help ensure the program fails safely.  Adopt A Secure Coding Standard – Standards like ERR-002-CPP ensure that our software performs as expected and prevents security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-ERR51** |  |
| CodeSonar | 9.1p0 | **LANG.STRUCT.UCTCH PARSE.MBDH** | Masked by handler Masked by default handler |
| Helix QAC | 2025.2 | **C++4035, C++4036, C++4037** |  |
| Klocwork | 2025.2 | **MISRA.CATCH.ALL** |  |
| LDRA tool suite | 9.7.1 | **527 S** | Partially implemented |
| Parasoft C/C++test | 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 | R2024b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| Security Reviewer - Static Reviewer | 6.02 | **C35** | Fully implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Random Number Generation | [MSC-001-CPP] | Do not use std::rand() to generate random numbers as it is not sufficiently random |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, std::rand() is used to generate random numbers for an idea. std::rand() is not sufficiently random and in some cases can be predictable. |
| #include <cstdlib>  #include <string>  void f() {  std::string id(“ID”);  id += std::to\_string(std::rand() % 10000);  } |

| **Compliant Code** |
| --- |
| In this compliant code example, different functions are used for random number generation which are sufficiently random and cannot be predicted. This consists of using two sources of randomness: the engine and the distribution. |
| #include <random>  #include <string>  void f() {  std::string id(“ID”);  std::uniform\_int\_distribution<int> distribution(0, 10000);  std::random\_device rd;  std::mt19937 engine(rd());  id += std::to\_string(distribution(engine));  } |

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

| **Principles(s):** Architect and Design for Security Policies – Choosing the correct implementation of random number generation is something that occurs during the design process.  Defense in Depth – Secure random number generation is one element of security that contributes to the overall security of the software.  Use Effective Quality Assurance Techniques – The use of std::rand() should be detected during the quality assurance process by static analysis tools or during code reviews. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **bad-function (AUTOSAR.26.5.1A)** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-MSC50** |  |
| Clang | 4.0 (prerelease) | cert-msc50-cpp | Checked by clang-tidy |
| CodeSonar | 9.1p0 | **BADFUNC.RANDOM.RAND** | Use of rand |
| ECLAIR | 1.2 | **CC2.MSC30** | Fully implemented |
| Helix QAC | 2025.2 | **C++5028** |  |
| Klocwork | 2025.2 | **CERT.MSC.STD\_RAND\_CALL** |  |
| LDRA tool suite | 9.7.1 | **44 S** | Enhanced Enforcement |
| Parasoft C/C++test | 2024.2 | **CERT\_CPP-MSC50-a** | Do not use the rand() function for generating pseudorandom numbers |
| Polyspace Bug Finder | R2024b | CERT C++: MSC50-CPP | Checks for use of vulnerable pseudo-random number generator (rule partially covered) |
| RuleChecker | 22.10 | **bad-function (AUTOSAR.26.5.1A)** | Fully checked |
| Security Reviewer - Static Reviewer | 6.02 | **RTOS\_07** | Fully implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| One Definition Rule | [DCL-001-CPP] | Ensure program does not contain multiple definition for non-inline functions and variables. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two different files (a.cpp and b.cpp) define a class with the same name with different definitions. |
| // a.cpp  struct S {  int a;  };  // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| In this compliant code example, the class S was stored in a header file and referenced by both files (a.cpp and b.cpp). |
| // S.h  struct S {  int a;  };  // a.cpp  #include “S.h”  // b.cpp  #include “S.h” |

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

| **Principles(s):** Heed Compiler Warnings – Violations of this coding standard are likely to be caught by the compiler.  Use Effective Quality Assurance Techniques – Good quality assurance techniques like unit testing, static analysis, and code reviews can be used to detect violations of this coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **type-compatibility definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading type-file-spreading** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL60** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **LANG.STRUCT.DEF.FDH LANG.STRUCT.DEF.ODH** | Function defined in header file Object defined in header file |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2025.2 | **C++1067, C++1509, C++1510** | [Insert text.] |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **286 S, 287 S** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-DCL60-a** | The One Definition Rule shall not be violated |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: DCL60-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl60cpp.html) | Checks for inline constraints not respected (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **type-compatibility definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading type-file-spreading** | Partially checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Do not read uninitialized memory | [EXP-001-CPP] | Unexpected behavior can occur if values are read before being initialized. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the variable i is printed before being initialized with a value. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the variable i is initialized with 0 before being printed. |
| #include <iostream>  void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** Heed Compiler Warnings – Modern compilers should issue warnings when uninitialized variables are used.  Use Effective Quality Assurance Techniques – Quality assurance techniques like static analysis, unit testing, and code reviews should be used to detect violations of this coding standard.  Architect and Design for Security Policies – Proper software design can help avoid this scenario. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **uninitialized-read** | Partially checked |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -Wuninitialized clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.1p0 | **LANG.STRUCT.RPL LANG.MEM.UVAR** | Return pointer to local Uninitialized variable |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2025.2 | **DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2025.2 | **UNINIT.CTOR.MIGHT UNINIT.CTOR.MUST UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST UNINIT.STACK.ARRAY.MIGHT UNINIT.STACK.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST UNINIT.STACK.MIGHT UNINIT.STACK.MUST** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **53 D, 69 D, 631 S, 652 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-EXP53-a** | Avoid use before initialization |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: EXP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp53cpp.html) | Checks for:   * Non-initialized variable * Non-initialized pointer   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.38 | [**V546**](https://pvs-studio.com/en/docs/warnings/v546/)**,**[**V573**](https://pvs-studio.com/en/docs/warnings/v573/)**,**[V614](https://pvs-studio.com/en/docs/warnings/v614/), [V670](https://pvs-studio.com/en/docs/warnings/v670/), [V679](https://pvs-studio.com/en/docs/warnings/v679/), [V730](https://pvs-studio.com/en/docs/warnings/v730/),[**V788**](https://pvs-studio.com/en/docs/warnings/v788/)**,**[V1007](https://pvs-studio.com/en/docs/warnings/v1007/), [V1050](https://pvs-studio.com/en/docs/warnings/v1050/) |  |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **uninitialized-read** | Partially checked |
| [Security Reviewer - Static Reviewer](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Security+Reviewer+-+Static+Reviewer) | 6.02 | **C54 C55 C56 C57 C58 C59 C60 C61 C62 C63** | Fully implemented |

### 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



Automation can be applied throughout the existing DevOps process and infrastructure to enforce our security standards. In the Assess and Plan and Design phases, we will plan how we will use automation to ensure the security of the software. This includes making decisions on which automation tools we will be using in the project and further refine our security requirements. In the Build phase, we will implement these automation tools into our project. During the Verify and Test phase, automation tools will be used to look for violations of the coding standards, verify required security functionality, and scan for other potential vulnerabilities before the code moves to production.

In the production cycle, automation continues to play an important role. During the Transition and Health Check phase, we can utilize automated tools to manage configurations and deploy our code to ensure that our production environments are securely configured and meet security requirements. In the Monitor and Detect phase, we will utilize automated threat detection, logging, and alerting tools to monitor our systems for potential threats. In the Respond phase, we can use automated tools to perform defensive actions like revoking credentials if malicious activity is suspected. During the Maintain and Stabilize phase, we can continue to use automated tools to scan for potential new vulnerabilities. Overall, automation plays an important role in the various phases of the DevOps process and helps to enhance the security of our software.

### 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 |
| --- | --- | --- | --- | --- | --- |
| INT-001-CPP | Medium | Unlikely | Low | High | L3 |
| INT-002-CPP | High | Likely | Medium | Medium | L2 |
| STR-001-CPP | High | Likely | Medium | Medium | L2 |
| STR-002-CPP | High | Likely | Medium | Medium | L2 |
| MEM-001-CPP | High | Likely | Low | Low | L1 |
| ERR-001-CPP | Medium | Unlikely | Medium | High | L3 |
| ERR-002-CPP | Low | Probable | Medium | Medium | L2 |
| MSC-001-CPP | Medium | Unlikely | Low | High | L3 |
| DCL-001-CPP | High | Unlikely | Medium | Medium | L2 |
| EXP-001-CPP | High | Probable | Medium | Low | L1 |

### 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 | Encryption at rest refers to protecting data stored on disk. For example, this could refer to data like backups or databases. Stored data should be encrypted using a secure encryption algorithm like AES. |
| Encryption in flight | Encryption in flight refers to protect data as it is being transmitted. An example of this is data being sent through API calls. Data in flight should be encrypted using TLS/SSL to secure all data being transmitted. |
| Encryption in use | Encryption in use refers to data that is being actively used. An example of this would be anything stored or processed in memory. To secure data in use, encryption techniques like homomorphic encryption or secure multi-party computation can be used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the process of verifying the identity of a user or device. Users and devices should verify their identity with login credentials, multi-factor authentication, and certificates. This policy helps to prevent unauthorized access by verifying users. |
| Authorization | Authorization refers to the process of checking what actions a user is allowed to take and what they can access. Users should only be allowed to access what is strictly necessary, following the principle of least privilege. Furthermore, we can implement role or policy-based access controls. These permissions would dictate whether users can perform aspects like making changes to the database, adding new users, or accessing certain files. |
| Accounting | Accounting refers to the process of logging user and system activity. This is important as it creates a history of activity that can be referred to when tracking user and system activity and can be used to investigate and identify potential security issues. Activity logged including changes or creation of users, changes to the database, files accessed or modified by users, user logins, and more. |

### 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 | 07/18/2025 | Completed Principles/Standards Draft | Dalton Rose |  |
| 1.2 | 08/06/2025 | Completed Principles/Standards | Dalton Rose |  |
| 1.3 | 08/08/2025 | Completed Summary of Risk Assessments & Automation | Dalton Rose |  |
| 1.4 | 08/09/2025 | Completed Encryption Policies | Dalton Rose |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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