**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 | Ensure that all the input data received by the system is validated to prevent any attacks such as SQL injection which will compromise the integrity and security of the application. |
| 1. Heed Compiler Warnings | Pay specific attention to compiler warnings during code compilation, as they can show that there are potential vulnerabilities in the system that can be exploited by attackers. |
| 1. Architect and Design for Security Policies | Incorporate security polices into the architectural and design phases of the SDLC to ensure that security is instilled from the ground up. |
| 1. Keep It Simple | Pursue to simplicity in design and implementation to minimize the complexity of a system, which will reduce the likelihood of introducing vulnerabilities and make it easier to maintain and secure. |
| 1. Default Deny | Use a default deny approach to access control, where access is denied by default and only granted to authorized entities, reducing the attack surface and limiting potential security breaches. |
| 1. Adhere to the Principle of Least Privilege | Grant users and process only the minimum level of access and permissions that are required to perform their duties. Which will then limit the potential impact of security incidents. |
| 1. Sanitize Data Sent to Other Systems | Clean and validate data before transmitting it to external or components to prevent injection attacks and ensure the data integrity and security. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls and mechanisms to protect against various types of threats and provide redundancy in case on layer fails. |
| 1. Use Effective Quality Assurance Techniques | Employ robust quality assurances practices, tests, and code reviews to identify and mitigate security vulnerabilities throughout the SDLC. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards and best practices to ensure consistency and adherence to security principles across all development teams to. Mitigate the risk of bringing new vulnerabilities. |

### 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** | [STD-001-CPP] | Data Type Consistency |

| **Noncompliant Code** |
| --- |
| Ensure consistent use of data types to prevent type-related vulnerabilities such as data corruption or overflows. |
| int x = 10;  float y = 5.5; |

| **Compliant Code** |
| --- |
| Ensuring that data types match their usage prevents any potential issues. |
| int x = 10;  float y = 5.5f; |

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

| **Principles(s):** Principle of least privilege: Enforcing consistent data types reduces the risk of type-related vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.06 | TypeCheck | Static analysis tool for detecting type-related issues. |
| CPPcheck | 2.6 | DataType | Static code analysis tool for detecting data-type inconsistencies. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-031-CPP] | Ensure Safe Data Conversions |

| **Noncompliant Code** |
| --- |
| The code converts an integer with a value of 300 to a char, which typically cannot represent values beyond 127 or 255, and this can result in data loss or misinterpretation. |
| int i = 300;  char c = i; |

| **Compliant Code** |
| --- |
| The code checks if the integer value falls within the range that can be represented by char, if it does it does its conversion, if not it will handle the error. |
| int i = 300;  if (i >= CHAR\_MIN && i <= CHAR\_MAX) {  char c = static\_cast<char>(i);  }  else {  //ERROR HANDLE  } |

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

| **Principles(s):** Ensuring safe data conversions aligns with the principle of fail-safe defaults by guaranteeing that any unexpected or incorrect data is handled securely, preventing potential vulnerabilities and maintaining system integrity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Atree | 24.04 | N/A | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| CodeSonar | 8.1p0 | 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 |
| Compass/Rose | N/a | N/A | Can detect violations of this rule. However, false warnings may be raised if limits.h is included |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-034-CPP] | Ensure String Safety and Correctness |

| **Noncompliant Code** |
| --- |
| The c\_str variable is used to traverse the character string containing the command line to be parsed. As characters are retrieved from this pointer, they are stored in a variable of type int. For implementations in which the char type is defined to have the same range, representation, and behavior as signed char, this value is sign-extended when assigned to the int variable. For character code 255 decimal (−1 in two's complement form), this sign extension results in the value −1 being assigned to the integer, which is indistinguishable from EOF |
| static int yy\_string\_get(void) {  register char \*c\_str;  register int c;    c\_str = bash\_input.location.string;  c = EOF;    /\* If the string doesn't exist or is empty, EOF found \*/  if (c\_str && \*c\_str) {  c = \*c\_str++;  bash\_input.location.string = c\_str;  }  return (c);  } |

| **Compliant Code** |
| --- |
| In this code, the result of the expression \*c\_str++ is cast to unsigned char before assignment to the int variable c. |
| **static** **int** yy\_string\_get(**void**) {  **register** **char** \*c\_str;  **register** **int** c;      c\_str = bash\_input.location.string;    c = EOF;      /\* If the string doesn't exist or is empty, EOF found \*/  **if** (c\_str && \*c\_str) {      /\* Cast to unsigned type \*/      c = (unsigned **char**)\*c\_str++;        bash\_input.location.string = c\_str;    }  **return** (c);  } |

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

| **Principles(s** **Ensuring string safety and correctness aligns with the principle of input validation and output encoding by verifying that all string inputs are properly checked and sanitized, preventing injection attacks and ensuring data integrity.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhause Suite | 7.2.0 | CertC-ERR34 | Fully Implemented |
| Astree | 24.04 | Char-sign-conversion | Fully Checked |
| CodeSonar | 8.1p) | Misc.NEGCHAR | Negative Character Value |
| ÉCLAIR | 1.2 | CC2.STR34 | Fully Implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-000-CPP] | SQL Injection prevention |

| **Noncompliant Code** |
| --- |
| The noncompliant code demonstrates a vulnerability to SQL injection attacks due to the use of concatenated strings to build SQL queries without proper sanitization. |
| String username = getUsernameFromInput();  String password = getPasswordFromInput();  String sqlString = "SELECT \* FROM db\_user WHERE username='" + username + "' AND password='" + password + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString); |

| **Compliant Code** |
| --- |
| The compliant code uses parameterized queries with placeholders for user input, preventing SQL injection attacks. |
| String username = getUsernameFromInput();  String password = getPasswordFromInput();  String sqlString = "SELECT \* FROM db\_user WHERE username=? AND password=?";  PreparedStatement stmt = connection.prepareStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, password);  ResultSet rs = stmt.executeQuery(); |

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

| **Principles(s):** SQL injection prevention aligns with the input validation and output encoding principle by ensuring that all inputs are properly validated and sanitized before being used in SQL queries, preventing malicious code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 8.1.p0 | Java.IO.INJ.SQL | SQL Injection (JAVA) |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-032-CPP] | Concurrency |

| **Noncompliant Code** |
| --- |
| Adjacent bit-files may be stored in a single memory location, Consequently, modifying adjacent bit-fields in different threads is undefined behavior as shown below. |
| struct multi\_threaded\_flags {  unsigned int flag1 : 2;  unsigned int flag2 : 2;  };    struct multi\_threaded\_flags flags;    int thread1(void \*arg) {  flags.flag1 = 1;  return 0;  }    int thread2(void \*arg) {  flags.flag2 = 2;  return 0;  } |

| **Compliant Code** |
| --- |
| This code protects all accesses of the flags with a mutex, thereby preventing any data races. |
| #include <threads.h>    struct multi\_threaded\_flags {  unsigned int flag1 : 2;  unsigned int flag2 : 2;  };    struct mtf\_mutex {  struct multi\_threaded\_flags s;  mtx\_t mutex;  };    struct mtf\_mutex flags;    int thread1(void \*arg) {  if (thrd\_success != mtx\_lock(&flags.mutex)) {  /\* Handle error \*/  }  flags.s.flag1 = 1;  if (thrd\_success != mtx\_unlock(&flags.mutex)) {  /\* Handle error \*/  }  return 0;  }    int thread2(void \*arg) {  if (thrd\_success != mtx\_lock(&flags.mutex)) {  /\* Handle error \*/  }  flags.s.flag2 = 2;  if (thrd\_success != mtx\_unlock(&flags.mutex)) {  /\* Handle error \*/  }  return 0;  } |

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

| **Principles(s):** Concurrency aligns with the least privilege principle by ensuring threads or processes have minimal permissions, reducing access violations and security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | read\_data\_race  write\_data\_race | Supported by sound analysis (data race alarm) |
| Axivion Bauhause Suite | 7.2.0 | CertC-CON32 | N/A |
| CodeSonar | 8.1p) | CONCURRENCY.DATARACE  CONCURRENCY.MAA | Data race  Multiple Accesses of Atomic |
| Coverity | 2017.07 | MISSING\_LOCK | Partially implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-003-CPP] | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| This code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions involving only using constant expressions, a preprocessor conditional statement may be used, as in this solution. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** Using a static assertion to test the value of a constant expression aligns with the fail-safe defaults principle by ensuring that critical assumptions are verified at compile time, preventing potential run-time errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axion Bauhaus Suite | 7.2.0 | CertC-DCL-3 | N/A |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.1p0 | (Customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/Rose | N/A | N/A | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-060-CPP] | exception objects must be no throw copy constructors |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an exception of type S is thrown in f(). However, because S has a std::string data member, and the copy constructor for std::string is not declared noexcept, the implicitly-defined copy constructor for S is also not declared to be noexcept. In low-memory situations, the copy constructor for std::string may be unable to allocate sufficient memory to complete the copy operation, resulting in a std::bad\_alloc exception being thrown. |
| #include <exception>  #include <string>    class S : public std::exception {  std::string m;  public:  S(const char \*msg) : m(msg) {}    const char \*what() const noexcept override {  return m.c\_str();  }  };    void g() {  // If some condition doesn't hold...  throw S("Condition did not hold");  }    void f() {  try {  g();  } catch (S &s) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the type of the exception object can inherit from std::runtime\_error, or that type can be used directly. Unlike std::string, a std::runtime\_error object is required to correctly handle an arbitrary-length error message that is exception safe and guarantees the copy constructor will not throw a exception |
| #include <stdexcept>  #include <type\_traits>    struct S : std::runtime\_error {  S(const char \*msg) : std::runtime\_error(msg) {}  };    static\_assert(std::is\_nothrow\_copy\_constructible<S>::value,  "S must be nothrow copy constructible");    void g() {  // If some condition doesn't hold...  throw S("Condition did not hold");  }    void f() {  try {  g();  } catch (S &s) {  // Handle error  }  } |

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

| **Principles(s):** Exception objects with no-throw copy constructors support robust exception handling by preventing additional exceptions during propagation, ensuring program stability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-err60-cpp | Checked by clang-tidy |
| Helix QAC | 2024.1 | C++3508 | N/A |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR60-a  CERT\_CPP-ERR60-b | Exception objects must be nothrow copy constructible  An explicitly declared copy constructor for a class that inherits from 'std::exception' should have a non-throwing exception specification |
| Polyspace Bug Finder | R2024a | CERT C++: ERR60-CPP | Checks for throwing exception object in copy constructor (rule fully covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-050-CPP] | Do not invoke virtual functions form constructors or destructors |

| **Noncompliant Code** |
| --- |
| [In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| struct B {  B() { seize(); }  virtual ~B() { release(); }    protected:  virtual void seize();  virtual void release();  };    struct D : B {  virtual ~D() = default;    protected:  void seize() override {  B::seize();  // Get derived resources...  }    void release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources. |
| class B {  void seize\_mine();  void release\_mine();    public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }    protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };    class D : public B {  void seize\_mine();  void release\_mine();    public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }    protected:  void seize() override {  B::seize();  seize\_mine();  }    void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):** Avoiding invocation of virtual functions from constructors or destructors ensures predictable behavior during object initialization and destruction, adhering to best practices in object-oriented design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | virtual-call-in-constructor  invalid\_function\_pointer | Fully Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP50 | N/A |
| Clang | 8.1p0 | clang-analyzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Helic QAC | 2024.1 | C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282 | N/A |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output (FIO) | [STD-052-CPP] | Do not alternately input and output from a file stream without an intervening positioning call. |

| **Noncompliant Code** |
| --- |
| his noncompliant code example appends data to the end of a file and then reads from the same file. However, because there is no intervening positioning call between the formatted output and input calls, the behavior is undefined. |
| include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Output some data";  std::string str;  file >> str;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::basic\_istream<T>::seekg() function is called between the output and input, eliminating the undefined behavior. |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Output some data";    std::string str;  file.seekg(0, std::ios::beg);  file >> str;  } |

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

| **Principles(s):** Avoid alternating input and output operations on a file stream without an intervening positioning call to ensure orderly and predictable file access, adhering to best practices in file handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-FIO50 | N/A |
| CodeSonar | 8.1p0 | IO.IOWOP  IO.OIWOP | Input After Output Without Positioning  Output After Input Without Positioning |
| Helix QAC | 2024.1 | DF4711, DF4712, DF4713 | N/A |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-FIO50-a | Do not alternately input and output from a stream without an intervening flush or positioning call |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers (CTR) | [STD-053-CPP] | Use Valid iterator ranges |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, std::for\_each() compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results 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 this compliant solution, the iterator values passed to std::for\_each() are passed 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; });  } |

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

| **Principles(s):** Iterator Validity - Using valid iterator ranges ensures that iterators remain within valid bounds when accessing elements in containers, preventing undefined behavior and maintaining program reliability and correctness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Overflow\_upon\_dereference | N/A |
| CodeSonar | 8.1p0 | LANG.MEM.BO | Buffer overrun |
| Helix QAC | 2024.1 | C++3902 | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |
| Polyspace Bug Finder | R2024a | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered). |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

**Explanation:** To enhance security and compliance in Green Pace's DevOps framework, automation will be integrated carefully across key phases of the DevSecOps pipeline. This includes embedding security requirements early in the planning phase, using IDE plugins for real-time code analysis, implementing robust testing and vulnerability scanning during verification, and deploying automated monitoring and response mechanisms in production. By making some changes to the existing DevOps process to include these automated security measures, Green Pace aims to enforce compliance to defined standards consistently. This approach will not only strengthen defense against potential threats but also ensures continuous improvement through proactive identification and mitigation of security risks across all stages of software development and deployment.

### 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 | High | 2 |
| STD-031-CPP | High | Probable | High | Low | 2 |
| STD-034-CPP | Medium | Probable | Medium | High | 2 |
| STD-000-CPP | High | Likely | Medium | High | 1 |
| STD-032-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-003-CPP | Low | Unlikely | High | Low | 3 |
| STD-060-CPP | Low | Probable | Medium | Low | 3 |
| STD-050-CPP | Low | Unlikely | Medium | Low | 3 |
| STD-51-CPP | Low | Unlikely | Medium | Medium | 2 |
| STD-53-CPP | High | Probable | High | Medium | 2 |

### 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 | Encrypting data stored in databases and files to prevent unauthorized access if physical media are compromised, ensuring data confidentiality at rest. |
| Encryption in flight | Encrypting data during transmission over networks to prevent interception and maintain data confidentiality and integrity during transit. |
| Encryption in use | Encrypting sensitive data within applications or systems to protect it from unauthorized access during processing, ensuring data security throughout its lifecycle. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying the identity of users or systems to prevent unauthorized access and ensure only legitimate entities gain entry to systems and data. |
| Authorization | Determining and enforcing what actions and resources authenticated users or systems are allowed to access, based on their roles and permissions, to prevent unauthorized activities. |
| Accounting | Tracking and recording user activities, access attempts, and data modifications to maintain visibility, support auditing, and facilitate compliance with regulatory requirements. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.5 | 6/15/2024 | Revised Template | Noah Lane Garza |  |
| [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 |