**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 | Validating all input data received by a system is essential to ensure that it is of the expected format and within acceptable ranges, as highlighted in this principle. By validating the input data, it is possible to reduce potential vulnerabilities such as buffer overflows or SQL injection attacks. |
| 1. Heed Compiler Warnings | During code compilation, compiler warnings are produced to notify developers of potential issues or unsafe coding practices. Adhering to this principle means paying attention to these warnings and dealing with them appropriately, as they may indicate vulnerabilities that could be exploited. |
| 1. Architect and Design for Security Policies | This principle highlights the need to integrate safety policies into the architecture and design of a system from the outset. Identifying and addressing potential vulnerabilities early on can be achieved by analyzing security requirements and implementing appropriate measures during the design phase. |
| 1. Keep It Simple | Simplicity is a fundamental principle in security. By keeping systems and code as simple as possible, the attack area is minimized, making it more difficult for attackers to exploit vulnerabilities. Complex systems tend to present more potential vulnerabilities and are harder to securely effectively. |
| 1. Default Deny | This principle advocates for a "default deny" approach, which requires permission to access system resources or sensitive data to be given only if explicitly authorized. The risk of unauthorized access is minimized by this method, ensuring that only necessary permissions are granted to users and processes. |
| 1. Adhere to the Principle of Least Privilege | The principle of minimum privileges stipulates that users and processes should have only the minimum privileges necessary for the performance of their tasks. In accordance with this principle, the impact of a potential security breach is reduced, as attackers will have limited access and control over the system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data when transacting with external entities or interacting with them is essential for preventing attacks like cross-site scripting (XSS) or SQL injection. The process of sanitizing data involves validating and cleaning input data to remove any malicious or unexpected content. |
| 1. Practice Defense in Depth | Defensive in depth is a strategy that involves the implementation of multiple layers of security controls to protect against different types of attacks. This principle suggests that the use of a single safety measure is inadequate, and that a combination of preventive, detection and reactive controls should be implemented to create a robust safety posture. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques, such as comprehensive code reviews, vulnerability analysis and penetration testing, are essential to identifying system security weaknesses and vulnerabilities. By utilizing effective quality assurance practices, potential safety deficiencies can be identified and resolved before deployment. |
| 1. Adopt a Secure Coding Standard | Adherence to a secure encoding standard is essential to ensure that developers adhere to best practices and avoid common programming errors that may introduce vulnerabilities. Promote the development of secure software by providing guidelines on secure coding practices, input validation, error handling, and other security-related aspects through secure coding standards. |

### 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] | STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string.  Passing a character sequence or wide character sequence that is not null-terminated to such a function can result in accessing memory that is outside the bounds of the object. |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant because the character sequence c\_str will not be null-terminated when passed as an argument to printf(). (See STR11-C. Do not specify the bound of a character array initialized with a string literal on how to properly initialize character arrays.) |
| #include <stdio.h>    void func(void) {  char c\_str[3] = "abc";  printf("%s\n", c\_str);  } |

| **Compliant Code** |
| --- |
| This compliant solution does not specify the bound of the character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character. |
| #include <stdio.h>    void func(void) {  char c\_str[] = "abc";  printf("%s\n", c\_str);  } |

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

| **Principles(s):** Validate Input Data - When validating input data, it is essential to ensure that any character sequence expected to be a string is correctly null-terminated. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.04 | wide-narrow-string-cast  wide-narrow-string-cast-implicit | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR38 | Fully implemented |
| CodeSonar | 7.4p0 | LANG.MEM.BO  LANG.MEM.TBA | Buffer Overrun  Tainted Buffer Access |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR38-a | Do not confuse narrow and wide character strings and functions |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | DCL59-CPP. Do not define an unnamed namespace in a header file.  Due to default internal linkage, each translation unit will define its own unique instance of members of the unnamed namespace that are ODR-used within that translation unit. This can cause unexpected results, bloat the resulting executable, or inadvertently trigger undefined behavior due to one-definition rule (ODR) violations. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the variable v is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of v and then assigns a new value into it. However, because v is defined within an unnamed namespace, each translation unit operates on its own instance of v, resulting in unexpected output. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    namespace {  int v;  }    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  #include <iostream>    void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }    // b.cpp  #include "a.h"  #include <iostream>    void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }    int main() {  extern void f();  f(); // Prints v, sets it to 42  g(); // Prints v, sets it to 100  f();  g();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, v is defined in only one translation unit but is externally visible to all translation units, resulting in the expected behavior. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    extern int v;    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  #include <iostream>    int v; // Definition of global variable v    void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }    // b.cpp  #include "a.h"  #include <iostream>    void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }    int main() {  extern void f();  f(); // Prints v, sets it to 42  g(); // Prints v, sets it to 100  f(); // Prints v, sets it back to 42  g(); // Prints v, sets it back to 100  } |

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

| **Principles(s):** Do not define an unnamed namespace in a header file - defining an unnamed namespace in a header file can lead to unintended consequences due to the One Definition Rule (ODR) violations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | unnamed-namespace-header | Fully checked |
| Clang | 3.9 | cert-dcl59-cpp | Checked by clang-tidy |
| LDRA tool suite | 9.7.1 | 286 S, 512 S | Fully implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL59-a | There shall be no unnamed namespaces in header files |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator.  Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a 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):** Use Effective Quality Assurance Techniques - Effective quality assurance techniques, including thorough testing and code reviews, can help identify instances where string buffers are not large enough to hold the data and null terminator. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Polyspace Bug Finder | R2023a | 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 |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer.  Calling std::unique\_ptr::release() will relinquish ownership of the managed pointer value. Destruction of, move assignment of, or calling std::unique\_ptr::reset() on a std::unique\_ptr object will also relinquish ownership of the managed pointer value, but results in destruction of the managed pointer value. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability. |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::shared\_ptr objects are related to one another through copy construction. When the local, automatic variable p2 is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local variable. |
| #include <memory>    void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):** Adhere to the Principle of Least Privilege - Ownership of the resource should be clearly defined and transferred between smart pointers as needed using move semantics or other appropriate techniques. This ensures that each smart pointer has exclusive control over its resource, preventing conflicts and improving resource management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM56 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| Polyspace Bug Finder | R2023a | CERT C++: MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | MEM51-CPP. Properly deallocate dynamically allocated resources.  Passing a pointer value to an inappropriate deallocation function can result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two allocations are attempted within the same try block, and if either fails, the catch handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for i1 may result in passing ::operator delete() a value (in i2) that was not previously returned by a call to ::operator new(), resulting in undefined behavior. |
| #include <new>    void f() {  int \*i1, \*i2;  try {  i1 = new int;  i2 = new int;  } catch (std::bad\_alloc &) {  delete i1;  delete i2;  }  } |

| **Compliant Code** |
| --- |
| This compliant solution initializes both pointer values to nullptr, which is a valid value to pass to ::operator delete(). |
| #include <new>    void f() {  int \*i1 = nullptr, \*i2 = nullptr;  try {  i1 = new int;  i2 = new int;  } catch (std::bad\_alloc &) {  delete i1;  delete i2;  }  } |

**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, such as thorough testing and code reviews, are essential to identify instances where dynamically allocated resources are not deallocated correctly. Code reviews can ensure that proper deallocation functions are called for each dynamically allocated resource, and testing can help catch memory-related issues like leaks or double deallocations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 7.4p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |
| Polyspace Bug Finder | R2023a | CERT C++: MEM51-CPP | Checks for:  Invalid deletion of pointer  Invalid free of pointer  Deallocation of previously deallocated pointer  Rule partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | DCL03-CPP.Use a static assertion to test the value of a constant expression.  Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code that uses this structure to behave correctly. |
| struct timer {  uint8\_t MODE;  uint32\_t DATA;  uint32\_t COUNT;  };    int func(void) {  assert(offsetof(timer, DATA) == 4);  } |

| **Compliant Code** |
| --- |
| This compliant solution mimics the behavior of static\_assert in a portable manner: |
| #define JOIN(x, y) JOIN\_AGAIN(x, y)  #define JOIN\_AGAIN(x, y) x ## y    #define static\_assert(e) \  typedef char JOIN(assertion\_failed\_at\_line\_, \_\_LINE\_\_) [(e) ? 1 : -1]    struct timer {  uint8\_t MODE;  uint32\_t DATA;  uint32\_t COUNT;  };    static\_assert(offsetof(struct timer, DATA) == 4); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques - By employing static assertions to test constant expressions, developers can verify critical assumptions and constraints in the codebase. This helps in eliminating potential bugs, security vulnerabilities, or undefined behavior related to incorrect assumptions about constant values, data types, or other compile-time characteristics. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| Compass/ROSE |  |  | 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 |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | ERR51-CPP. Handle all exceptions.  As per ERR50-CPP-EX1, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code 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 this compliant solution, the main entry point 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  }  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques - The "Use Effective Quality Assurance Techniques" principle emphasizes the importance of applying rigorous testing, code reviews, and quality assurance measures to identify and fix potential defects and security vulnerabilities in software development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| CodeSonar | 7.4p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++test | 2023.1 | 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 | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | [STD-008-CPP] | FIO46-C. Do not access a closed file.  Using the value of a pointer to a FILE object after the associated file is closed is undefined behavior. (See undefined behavior 148.) Programs that close the standard streams (especially stdout but also stderr and stdin) must be careful not to use these streams in subsequent function calls, particularly those that implicitly operate on them. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the stdout stream is used after it is closed. |
| #include <stdio.h>    int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }    printf("stdout successfully closed.\n");  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, stdout is not used again after it is closed. This must remain true for the remainder of the program, or stdout must be assigned the address of an open file object. |
| #include <stdio.h>    int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }    fputs("stdout successfully closed.", stderr);  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, such as thorough testing and code reviews, play a crucial role in detecting and preventing such issues. Test cases can be designed to verify that files are correctly opened, used, and closed without any attempts to access them after closure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.04 |  | Supported |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Implemented |
| LDRA tool suite | 9.7.1 | 48 D | Partially implemented |
| PC-lint Plus | 1.4 | 2471 | Fully supported |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Arrays | [STD-009-CPP] | ARR36-C. Do not subtract or compare two pointers that do not refer to the same array.  When two pointers are subtracted, both must point to elements of the same array object or just one past the last element of the array object, the result is the difference of the subscripts of the two array elements. Otherwise, the operation is undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, pointer subtraction is used to determine how many free elements are left in the nums array. |
| #include <stddef.h>    enum { SIZE = 32 };    void func(void) {  int nums[SIZE];  int end;  int \*next\_num\_ptr = nums;  size\_t free\_elements;    /\* Increment next\_num\_ptr as array fills \*/    free\_elements = &end - next\_num\_ptr;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the number of free elements is computed by subtracting next\_num\_ptr from the address of the pointer past the nums array. While this pointer may not be dereferenced, it may be used in pointer arithmetic. |
| #include <stddef.h>  enum { SIZE = 32 };    void func(void) {  int nums[SIZE];  int \*next\_num\_ptr = nums;  size\_t free\_elements;    /\* Increment next\_num\_ptr as array fills \*/    free\_elements = &(nums[SIZE]) - next\_num\_ptr;  } |

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

| **Principles(s):** Heed Compiler Warnings - Compiler warnings often help programmers identify potential issues or undefined behavior in their code. By heeding compiler warnings, developers can catch these problems early in the development process and address them to ensure the code's correctness and safety. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.04 | pointer-subtraction | Partially checked |
| CodeSonar | 7.4p0 | LANG.STRUCT.CUP  LANG.STRUCT.SUP | Comparison of Unrelated Pointers  Subtraction of Unrelated Pointers |
| Coverity | 2017.07 | MISRA C 2004 17.2  MISRA C 2004 17.3  MISRA C 2012 18.2  MISRA C 2012 18.3 | Implemented |
| LDRA tool suite | 9.7.1 | 437 S, 438 S | Fully implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Environment | [STD-010-CPP] | ENV30-C. Do not modify the object referenced by the return value of certain functions.  Some functions return a pointer to an object that cannot be modified without causing undefined behavior. These functions include getenv(), setlocale(), localeconv(), asctime(), and strerror(). In such cases, the function call results must be treated as being const-qualified. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example modifies the string returned by getenv() by replacing all double quotation marks (") with underscores (\_). |
| #include <stdlib.h>    void trstr(char \*c\_str, char orig, char rep) {  while (\*c\_str != '\0') {  if (\*c\_str == orig) {  \*c\_str = rep;  }  ++c\_str;  }  }    void func(void) {  char \*env = getenv("TEST\_ENV");  if (env == NULL) {  /\* Handle error \*/  }  trstr(env,'"', '\_');  } |

| **Compliant Code** |
| --- |
| If the programmer does not intend to modify the environment, this compliant solution demonstrates how to modify a copy of the return value. |
| #include <stdlib.h>  #include <string.h>    void trstr(char \*c\_str, char orig, char rep) {  while (\*c\_str != '\0') {  if (\*c\_str == orig) {  \*c\_str = rep;  }  ++c\_str;  }  }    void func(void) {  const char \*env;  char \*copy\_of\_env;    env = getenv("TEST\_ENV");  if (env == NULL) {  /\* Handle error \*/  }    copy\_of\_env = (char \*)malloc(strlen(env) + 1);  if (copy\_of\_env == NULL) {  /\* Handle error \*/  }    strcpy(copy\_of\_env, env);  trstr(copy\_of\_env,'"', '\_');  /\* ... \*/  free(copy\_of\_env);  } |

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

| **Principles(s):** Default Deny - By adhering to the "Default Deny" principle, you ensure that you are cautious about modifying data returned from functions like getenv(), setlocale(), localeconv(), asctime(), and strerror(), which are documented to return pointers to read-only data or data that should not be modified. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | BADFUNC.GETENV  LANG.STRUCT.RPNTC | Use of getenv  Returned Pointer Not Treated as const |
| Compass/ROSE |  |  | Can detect violations of this rule. In particular, it ensures that the result of getenv() is stored in a const variable |
| Parasoft C/C++test | 2023.1 | CERT\_C-ENV30-a | The pointers returned by the Standard Library functions 'localeconv', 'getenv', 'setlocale' or, 'strerror' shall only be used as if they have pointer to const-qualified type |
| Polyspace Bug Finder | R2023a | CERT C: Rule ENV30-C | Checks for modification of internal buffer returned from nonreentrant standard function (rule fully 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.

[Insert your written explanations here.]

### 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 |
| STR32-C | High | Likely | Low | P27 | L1 |
| DCL59-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STR50-CPP | High | Likely | Medium | P18 | L1 |
| MEM56-CPP | High | Likely | Medium | P18 | L1 |
| MEM51-CPP | High | Likely | Medium | P18 | L1 |
| DCL03-CPP | Low | Unlikely | High | P1 | L3 |
| ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| FI046-C | Medium | Unlikely | Medium | P4 | L3 |
| ARR36-C | Medium | Probable | Medium | P8 | L2 |
| ENV30-C | Low | Probable | Medium | P4 | L3 |

### 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 in rest | Encryption at rest involves encrypting data when it is stored or saved on a storage medium like hard drives, databases, or backup tapes. This policy is applied to all data at rest in storage devices or repositories and is essential for safeguarding data in the event of physical theft or unauthorized access to storage media.  To ensure data security, all sensitive and confidential information stored on company servers or devices must be encrypted using robust encryption algorithms. Access to encryption keys and decryption processes should be tightly controlled and restricted to authorized personnel only. Additionally, encryption keys must be securely stored to prevent unauthorized access. |
| Encryption at flight | Encryption in flight refers to the encryption process that takes place when data is being transmitted over networks or communication channels. This type of encryption is implemented to safeguard data from interception and eavesdropping during its transmission. Encryption in flight is commonly employed to secure data sent over the internet, intranets, or any other communication channel.  To maintain data confidentiality and integrity during transmission, it is mandatory that all data sent over public or untrusted networks be encrypted using industry-standard encryption protocols, such as TLS/SSL. This policy is applicable to all communication between internal systems and any data exchanged with external parties. Adherence to this policy ensures that sensitive information remains protected from unauthorized access while in transit. |
| Encryption in use | Encryption in use is a method that enables data to stay encrypted while it undergoes processing by applications or services. This type of encryption proves especially valuable when dealing with sensitive data that needs to be processed in untrusted environments, such as cloud services or third-party applications.  It is imperative that sensitive data be processed in encrypted form whenever feasible, utilizing secure applications and services that support encryption in use. Access to decrypted data must be restricted to the minimum level required to perform essential tasks. The encryption keys utilized during processing should be managed with the utmost security measures and disposed of once the processing is complete. By adhering to this policy, data confidentiality is maintained throughout its processing lifecycle, safeguarding it from potential breaches or unauthorized access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of validating the identity of a user or entity attempting to access a system or resource. Its purpose is to ensure that only legitimate and authorized users can gain access to specific information or perform certain actions within the system.  Before granting access to the organization's systems and data, all users must undergo authentication. To enhance security for sensitive systems and data, robust authentication methods like multi-factor authentication (MFA) should be implemented. Additionally, user logins and login attempts should be logged and continuously monitored to promptly identify and respond to any unauthorized access attempts. |
| Authorization | Authorization determines the actions or resources that a user can access after successful authentication. It defines the level of access granted to each user based on their role, and privileges, and adheres to the principle of least privilege.  Once authentication is successfully completed, users should only be granted access to resources and actions essential for their respective job roles. Fine-grained access control should be enforced through mechanisms like access control lists (ACLs) and role-based access control (RBAC). Any changes to user access permissions must be thoroughly audited and authorized by the appropriate personnel. |
| Accounting | Accounting involves tracking and recording user activities and system events. It creates a traceable record of actions performed by users and enables investigations in the event of security incidents or policy violations.  Regular logging and auditing of all user activities, database changes, additions of new users, user access levels, and files accessed by users are mandatory. Audit logs should be protected from tampering or unauthorized access. Routine reviews of audit logs should be conducted to promptly detect and respond to any suspicious or unauthorized activities. |

**\***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 |  |
| 2.0 | 08/05/2023 | Inserted data for principle, threat, and automation. Completed summary of risk assessment and policies of encryption and Tiple-A framework. | David Allen |  |
| [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 |