**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 | Input validation ensures that only properly formed data is entering the workflow in an information system. Validating input data prevents malformed data from entering the database and triggering malfunctions of various downstream components. Input validation should happen as early as possible in the dataflow. All external data that enters the system should be treated as “untrusted data”. Input validation should be applied on both syntactical and semantic level. Syntactic refers to correct syntax of structured fields. Semantic refers to correctness of values in the specific business context. |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for the compiler. Use static and dynamic analysis tools to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Develop software architecture and design software around established security policies. Software design and architecture should be implemented to enforce security policies. |
| 1. Keep It Simple | Avoid complex system configurations that introduce bugs and vulnerabilities. Keep the design simple and modular to so that they are easy to implement, update, and refactor. |
| 1. Default Deny | The system must always decide to either deny or permit requested access. Access decisions should be based on permissions to explicitly justify why a specific permission was granted to a particular user or group. Explicit configuration is preferred over framework or library defaults. |
| 1. Adhere to the Principle of Least Privilege | Least privilege is a security concept that refers to the principle of assigning users only the minimum privileges necessary to complete their job. During design, ensure trust boundaries are defined. Create tests that validate permissions mapped out in the design phase are being correctly enforced. Periodically review permissions in the system for “privilege creep”. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data passed through subsystems such as command shells or relational database components. Attackers may be able to invoke unused functionality in these components using SQL, command, or other types of injection attacks. |
| 1. Practice Defense in Depth | Assess and manage risk with multiple defense strategies across all the system’s layers. This will create layered fail-safes in case there is an unknown penetrable vulnerability or attack to the system. Defense in depth involves physical security, network security, application security, and data security. Implementing security strategies across the strata of system layers will reduce the risk of successful attacks and minimize damage caused by successful attacks. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance can be effective for identifying and eliminating vulnerabilities. Penetration testing and static analysis should be incorporated as a part of an effective quality assurance program. Independent security reviews and external audits can result in enhanced security of a system. |
| 1. Adopt a Secure Coding Standard | Develop a secure coding standard for your target development language and platform. |

#### Coding Standard 1

| **Coding Standard** | **Label** | **EXP51-CPP. Do not delete an array through a pointer of the incorrect type** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Deleting an array through a pointer to the incorrect type results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| An array of Derived objects is created and the pointer is stored in a Base \*. Despite Base::~Base() being declared virtual, it still results in undefined behavior. |
| **struct** Base {  **virtual** ~Base() = **default**;  };    **struct** Derived final : Base {};    **void** f() {     Base \*b = **new** Derived[10];     // ...  **delete** [] b;  } |

| **Compliant Code** |
| --- |
| The static type of b is Derived \*, which removes the undefined behavior when indexing into the array as well as when deleting the pointer. |
| **struct** Base {  **virtual** ~Base() = **default**;  };    **struct** Derived final : Base {};    **void** f() {     Derived \*b = **new** Derived[10];     // ...  **delete** [] b;  } |

| **Principles(s):** (4) Keep It Simple, (9) Use Effective Quality Assurance Techniques.  Deleting an array through a pointer to the incorrect type results in undefined behavior. Attempting to destroy an array of polymorphic objects through the incorrect static type is undefined behavior which can result in abnormal program execution and memory leaks. The coding standard relates to principle 4 and 9. Principle 4 requires that system configuration should be kept simple to avoid the introduction of bugs and vulnerabilities that result from complex configurations. Keeping the design simple and modular ensures that easy vulnerabilities such as the one in the non-compliant code mentioned in this standard do not creep into the code. These vulnerabilities can be easy to miss and are more difficult to detect using automated tools. Quality assurance techniques can be used to identify and eliminate vulnerabilities using static analysis tools to ensure that arrays are not treated polymorphically or the delete operator used to destroy the downcast object of a different type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -analyzer-checker=cplusplus | Checked with clang –cc1 or (preferably) scan-build |
| CodeSonar | 7.3p0 | ALLOC.TM | Type Mismatch |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-EXP51-a | Do not treat arrays polymorphically |
| Polyspace Bug Finder | R2023a | CERT C++:EXP51-CPP | Checks for delete operator used to destroy downcast object of different type |

#### Coding Standard 2

| **Coding Standard** | **Label** | **CTR50-CPP. Guarantee that container indices and iterators are within the valid range.** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensuring that array references are within the bounds of the array and vector integer indexes are within the bounds of the vector are the responsibility of the programmer. If not checked, the index values can be modified by untrusted sources. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, **int** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, std::**size\_t** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

| **Principles(s):** (1) Validate Input Data. (8) Practice Defense in Depth. (10). Adopt a Secure Coding Standard.  Using an invalid array or container index can result in an arbitrary memory overwrite or abnormal program termination. The coding standard applies to the first principle, 1. Validate Input Data, because the standard requires that that input validation occurs on the semantic level which refers to the correctness of values in the specific business context. Range checking values validates data to ensure that elements outside of the bounds of the container are not improperly overwritten. The coding standard relates to Principle 8 and 10 because input validation should happen as early as possible in the dataflow which provides a layer of security at the application and user interface level, and range-checking should apply to all containers within the coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Overflow\_upon\_dereference | [Insert text.] |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X. | Partially implemented. |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range. |
| Polyspace Bug Finder | R2023a | CERT C++: CTR50-CPP | Checks for:  Array access out of bounds, array access with tainted index, and point dereference with tainted offset. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **STR51-CPP. Do not attempt to create a std::string from a null pointer.** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | The std::basic\_string type uses the *traits* design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Passing a null pointer to these functions results in undefined behavior because it would result in dereferencing a null pointer. |

| **Noncompliant Code** |
| --- |
| The example code below shows a std::string object that is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** (1) Validate Input Data, (2) Heed Compiler Warnings, (3) Architect and Design for Security Policies, (9) Use Effective Quality Assurance Techniques.  Passing a null pointer to an std::string class object results in undefined behavior because it would result in dereferencing a null pointer, typically abnormal program termination or in some cases, can lead to the execution of arbitrary code. The coding standard is related to principle 1, 2, 3, and 9 because the code checks for null before the std::string object is created. Creating an std::string class object from a null pointer may result in a compiler warning which should serve as an indication that there is a potential issue with the code; however, the code can still run if the warning goes unchecked. A check for a null pointer reference to a std::string object should be implemented in the software architecture and design which serves to validate input data to prevent arbitrary code execution. Using effective quality assurance techniques ensures that checking objects for null before creating string objects ensures that string objects are not created from null pointers. A null pointer check eliminates vulnerabilities that can result in arbitrary code execution that can eventually lead to a denial-of-service attack. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Assert\_failure |  |
| CodeSonar | 7.3p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023a | CERT C++:STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **FIO47-C. Use valid format strings** |
| --- | --- | --- |
| **SQL Injection** | STD-004-C | Do not supply an unknown or invalid conversion specification or an invalid combination of flag character, precision, length modifier, or conversion specifier to a formatted IO function. Do not provide a number or type of argument that does not match the argument type of the conversion specifier used in the format string. Mismatches between arguments and conversion specifications may result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The error\_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. Likewise, the error\_msg argument is incorrectly matched with the d specifier instead of the s specifier. |
| #include <stdio.h>    void func(void) {  const char \*error\_msg = "Resource not available to user.";  int error\_type = 3;  /\* ... \*/  printf("Error (type %s): %d\n", error\_type, error\_msg);  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications: |
| #include <stdio.h>    void func(void) {  const char \*error\_msg = "Resource not available to user.";  int error\_type = 3;  /\* ... \*/  printf("Error (type %d): %s\n", error\_type, error\_msg);    /\* ... \*/  } |

| **Principles(s):** (1) Validate Input Data, (5) Default Deny, (7) Sanitize Data Sent to Other Systems, (8) Practice Defense in Depth, (9) Use Effective Quality Assurance Techniques, (10) Adopt a Secure Coding Standard.  Incorrectly specified format strings can result in memory corruption or abnormal program termination. Mismatches between arguments and conversion specifications may result in undefined behavior. Behavior from mismatches between arguments and conversion specifications may possibly result in an access violation. Using valid format strings relates to input validation and data sanitization principles because format strings containing user input can enable attackers to execute malicious code to control its content in an I/O function allowing them to view memory content, check contents of the stack, or modify a memory location. This ensures that input data and data sent to other systems are sanitized before being passed. Using valid format strings applies the default deny, quality assurance, adopting secure coding standards, and a defense in depth principles by specifying compliant code practices to ensure correct format strings are applied in the source code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO47 | Fully implemented |
| CodeSonar | 7.3.p0 | IO.INJ.FMT  MISC.FMT  MISC.FMTTYPE | Format string injection  Format string  Format string type error |
| Coverity | 2017.07 | PW | Reports when number of arguments differs from the number of required arguments according to the format string. |
| LDRA tool suite | 9.7.1 | 486 S  589 S | Full implemented. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **MEM50-CPP. Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Evaluating a pointer into memory that has been deallocated by a memory management function is undefined behavior. Accessing pointers that have been deallocated by memory can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Variable s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| The dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

| **Principles(s):** (8) Practice Defense in Depth, (9) Use Effective Quality Assurance Techniques, (10) Adopt a Secure Coding Standard.  Reading previously dynamically allocated memory after it has been deallocated can lead to abnormal program termination and denial-of-service attacks. Writing memory that has been deallocated can lead to the execution of arbitrary code with the permissions of the vulnerable process. Accessing a dangling pointer can result in exploitable vulnerabilities. Memory must not be written to or read from once it is freed. The coding standard relates primarily to principle 9 and 10 because the error resulting from non-compliant code is a result of quality coding practices. Quality assurance techniques that can effectively identify the vulnerability through static code analysis can flag the code as non-compliant and the code can be easily rewritten to free the memory only after it is no longer required. The standard should be a part of a secure coding standard to eliminate memory access vulnerabilities related to deallocating freed memory only after it has been used. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.8 | Clang-analyzer-cplusplus.NewDelete  Clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule |
| Coverity | V7.5.0 | USE-AFTER\_FREE | Can detect specific instances, where memory is deallocated more than once or read/written to the target of a freed pointer. |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed. |
| Polyspace Bug Finder | R2023a | CERT C++:MEM50-CPP | Checks for:  Pointer access out of bounds, deallocation of previously deallocated pointer, use of previously freed pointer. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **DCL03-C. Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | STD-006-C | The runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. Runtime assertions are generally unsuitable for server programs or embedded systems. The static assertion method should be used instead which performs a compile-time check. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed. |
| #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 constant expressions, a preprocessor conditional statement may be used. Using #error directives allows for clear diagnostic messages. Because this approach evaluates assertions at compile time, there is no runtime penalty.  This portable compliant solution uses static\_assert: |
| 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  static\_assert(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**),                "Structure must not have any padding"); |

| **Principles(s):** (9) Use Effective Quality Assurance Techniques.  Runtime assertions are generally unsuitable for server programs or embedded systems because it is only useful for identifying incorrect assumptions and not for runtime error checking. Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. The coding standard relates to principle 9 because an assertion failure results in a meaningful and informative diagnostic error message. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Mis-static-assert | Checked by clang-tidy |
| CodeSonar | 7.3p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **ERR51-CPP. Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | The default terminate handler called by std::terminate() calls std::abort(), which abnormally terminates the process. When std::abort() is called, or if the implementation does not unwind the stack prior to calling std::terminate(), destructors for objects may not be called and external resources can be left in an indeterminate state which can lead to denial-of-service attacks. |

| **Noncompliant Code** |
| --- |
| 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** |
| --- |
| 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  }  } |

| **Principles(s):** (1) Validate input Data, (3) Architect and Design for Security Policies, (5) Default Deny, (7) Sanitize Dta Sent to Other Systems, (10) Adopt a Secure Coding Standard.  Allowing the application to abnormally terminate can lead to resources not being freed or closed. Abnormal termination is a frequent vector for denial-of-service attacks. All exceptions thrown by an application must be caught by a matching exception handler. The standard is related to principles 1, 3, 5, 7, and 10. Assume all input is malicious. Use an “accept known good” input validation strategy using a list of acceptable inputs and rejecting any input that does not strictly conform to specifications, or transform it into something that does. If the program must fail, ensure that it fails gracefully by ensuring that all exception handling is defined to avoid a call to std::terminate(). Use system limits to prevent resource exhaustion. Ensure that error messages only contain minimal details that are useful to the intended audience and no one else. Check the results of all functions that return a value and verify that the value is expected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Main-function-catch-all-early-cath-all | Partially checked |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++test | 2022.2 | 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 to that point. |
| Polyspace Bug Finder | R2023a | CERT C++:ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **INT50-CPP. Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| Integers | STD-008-CPP | Enumerations in C++ come in both scoped enumerations with a fixed underlying type and unscoped enumerations with a fixed or non-fixed underlying type. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. The arithmetic value being cast must be within the range of values the enumeration can represent to avoid operating on unspecified values. When dynamically checking for out-of-range values, checking must be performed before the cast expression. |

| **Noncompliant Code** |
| --- |
| The code attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

| **Principles(s):** (7) Sanitize Data Sent to Other Systems, (8) Practice Defense In Depth.  Iti is possible for unspecified values to result in a buffer overflow, leading to execution of arbitrary code by an attacker. The coding standard is related to principle 7 and 8 because it ensures a check to the range of enumeration elements before the cast expression. The range check sanitizes the data before casting to prevent the vulnerability. Because enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will result in data integrity violations rather than arbitrary code execution. To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. Implementing this coding standard is a part of a defense in depth approach to software security because it restricts enumeration casting to its range of available values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Astree | 22.10 | Cast-integer-to-enum | Partially checked |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration. |
| RuleChecker | 22.10 | Cat-integer-to-enum | Partially checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **STR53-CPP. Range check element access** |
| --- | --- | --- |
| Value Range Checking | STD-009-CPP | The std::string index operators const\_reference operator[](size\_type) const and reference operator[](size\_type) return the character stored at the specified position, pos. When pos >= size(), a reference to an object of type charT with value charT() is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the value returned by the call to get\_index() may be greater than the number of elements stored in the string, resulting in undefined behavior. |
| #include <string>    extern std::size\_t get\_index();    void f() {  std::string s("01234567");  s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value returned by get\_index() is within a valid range before calling operator[](). |
| #include <string>    extern std::size\_t get\_index();    void f() {  std::string s("01234567");  std::size\_t i = get\_index();  if (i < s.length()) {  s[i] = '1';  } else {  // Handle error  }  } |

| **Principles(s):** (3) Architect and Design for Security Policies, (5) Default deny, (6) Adhere to the Principle of Least Privilege, (8) Defense in Depth.  Unchecked element access can lead to out-of-bound reads and writes and write-anywhere exploits. These exploits can, in turn, lead to the execution of arbitrary code with the permissions of the vulnerable process. This coding standard applies to principles 3, 5, 6, and 8 because unchecked element access can result in undefined behavior and exploitable conditions. Restricting range access to elements is a coding standard that should be implemented in the architecture and design of security policies along with trust boundary definitions. Requesting access outside the range of elements of an object should always be denied. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Assert\_failure | [Insert text.] |
| CodeSonar | 7.3p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range. |
| Polyspace Bug Finder | R2023a | CERT C++:STR53-CPP | Checks for:  Array access out of bounds, array access with tainted index,  Pointer dereference with tained offset.  Rule partially covered. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **DCL53-CPP. Do not write syntactically ambiguous declarations** |
| --- | --- | --- |
| Declarations | STD-010-CPP | It is possible to devise syntax that can ambiguously be interpreted as either an expression statement or a declaration. With the advent of uniform initialization syntax using a braced-init-list, there is now syntax that unambiguously specifies a declaration instead of an expression statement. Declarations can also be disambiguated by using nonfunction-style casts, by initializing using =, or by removing extraneous parenthesis around the parameter name. |

| **Noncompliant Code** |
| --- |
| An attempt is made to declare a local variable, w, of type Widget while executing the default constructor. However, this declaration is syntactically ambiguous where the code could be either a declaration of a function pointer accepting no arguments and returning a Widget or a declaration of a local variable of type Widget. |
| #include <iostream>    struct Widget {  Widget() { std::cout << "Constructed" << std::endl; }  };    void f() {  Widget w();  } |

| **Compliant Code** |
| --- |
| The compliant code shows two equally compliant ways to write the declaration. The first way is to elide the parentheses after the variable declaration, which ensures the syntax is that of a variable declaration instead of a function declaration. The second way is to use a braced-init-list to direct-initialize the local variable. |
| #include <iostream>    struct Widget {  Widget() { std::cout << "Constructed" << std::endl; }  };    void f() {  Widget w1; // Elide the parentheses  Widget w2{}; // Use direct initialization  } |

| **Principles(s):** (3) Architect and Design for Security Policies, (9) Use Effective Quality Assurance Techniques, (10) Adopt a Secure Coding Standard.  Do not write a syntactically ambiguous declaration. With the advent of uniform initialization syntax using a braced-init-list, there is now syntax that unambiguously specifies a declaration instead of an expression statement. Declarations can also be disambiguated by using nonfunction-style casts, by initializing using =, or by removing extraneous parenthesis around the parameter name. The coding standard relates to principles 3, 9, and 10 because using syntactically ambiguous declarations should be disallowed in the software design and coding standards. Syntactically ambiguous declarations can lead to unexpected program execution. However, it is likely that rudimentary testing would uncover violations of this rule. Effective quality assurance techniques should be able to reveal violations of the standard using static code analysis and review. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b  CERT\_CPP-DCL53-c | Parameter names in function declarations should not be enclosed in parentheses.  Local variable names in variable declarations should not be enclosed in parentheses.  Avoid function declarations that are syntactically ambiguous. |
| Polyspace Bug Finder | R2023a | CERT C++: DCL53-CPP | Checks for declarations that can be confused between:  Function and object declaration.  Unnamed object or function parameter declaration.  Rule fully covered. |
| LDRA tool suite | 9.7.1 | 296 S | [Insert text.] |

### Defense-in-Depth Illustration

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



### Automation

Provide a written explanation using the image provided.



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.

Automation should take place in the pre-production cycle within the DevSecOps process. Automation enables security at scale and should be implemented through the entire security development lifecycle. In the assessment and planning phase, it is a great opportunity for the organization to provide a high-level overview of the threat landscape using threat modeling techniques as well as introduce security concepts and best practices through security training and education. Security training and education will ensure that developers adopt coding standards based on the 10 defined coding principles that will drive the software architecture and design. The design phase will implement these coding standards by restricting the design to adhere to the security policies. As the systems are built, automated static analysis tools and plugins can be used to scan code for vulnerabilities as it is written before testing and verification. Automated processes for managing open-source and third-party dependencies are essential to securing the software system. Code dependency checks such as the OWASP Dependency-Check can help ensure that code with known vulnerabilities is minimized. The dependency check should be used at certain intervals as vulnerabilities in dependencies are continuously discovered. During verification and testing, dynamic tests can be applied automatically to test the code at runtime for functionality and compliance with standards.

As the cycle moves into the production cycle of the SDL, additional security measures are required to observe the system for potential vulnerabilities and attacks. Monitoring the system in a live environment can be done by using automated security checks and security monitoring loops. Runtime application self-protection (RASP) automatically identifies and blocks inbound security threats in real-time acting as a reverse proxy that observes incoming attacks and enables the application to reconfigure automatically without human intervention in response to explicit conditions. Security monitoring uses analytics to instrument and monitor critical security-related metrics.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Medium | 2 | 3 |
| STD-002-CPP | High | Likely | High | 9 | 2 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CPP | High | Unlikely | Medium | 6 | 2 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Unlikely | High | 1 | 3 |
| STD-007-CPP | Low | Likely | Medium | 4 | 3 |
| STD-008-CPP | Medium | Unlikely | Medium | 4 | 3 |
| STD-009-CPP | High | Unlikely | Medium | 6 | 2 |
| STD-010-CPP | Low | Unlikely | Medium | 2 | 3 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Data is considered at rest when it resides on a storage device and is not actively being used or transferred. Encrypting data at rest means to encrypt stored data to prevent unauthorized access. Encrypting data at rest reduces the likelihood of data loss or theft in cases of data breaches, data leakages, and lost or stolen devices. If an unauthorized entity accesses encrypted data, the entity cannot access the raw data without the decryption key making it more difficult to access the data. Thus, encryption at rest protects stored data adding extra protection to a system’s data by securing the contents of the data even in the case of unauthorized access. |
| Encryption at flight | Encryption at flight encrypts data that is in transit. Encryption at flight occurs when the encrypted data is actively moving between devices and networks. Encryption at flight protects the data as it travels from one location to another. This type of encryption is important to prevent data leakage or breaches when data is being transported across a network.  Data in transit shall always be encrypted. |
| Encryption in use | Encryption in use refers to the encryption of data that is being used by a central processing unit such as updating, deleting, creating, and reading. Encryption of data in use is the most difficult stage of encryption but is vital to reducing data breach risks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication identifies a user and ensures the validity of the user’s credentials. User authentication ensures proper authorization to access a system is granted. Users must provide proper login information with a minimum of a username and password to access the system. Users that cannot provide proper access controls shall not be allowed to access the system. Multiple attempts to access the system with invalid credentials shall result in locking of the account to prevent potential unauthorized access to the system where credentials such as password can be changed only by a system administrator or other system user that has role-based privileges to change a user’s password. |
| Authorization | Authorization refers to the process of enforcing policies that govern the resources, services, and activities that a particular user is permitted to access and use. Users are assigned authorization levels that define their access to a network and associated resources. Permissions to make changes to the database must not be enabled for all users but only those authorized to do so such as system administrators. Access control must be defined using role-based access control that is determined upon creation of a new user account. Files accessed by users must be based on their role assigned to them upon account creation. |
| Accounting | Accounting measures the resources users consume during access to a network or application, logging session statistics and user information such as session duration, and data sent and received. Accounting ensures that an audit will enable administrators to login and view actions performed, identifying activities by users as well as the time the system component was accessed. User logins shall be monitored and accounted for. Files accessed by users shall be monitored to ensure that only authorized users access files authorized by their role. File access attempts by users without access privileges or with restricted access privileges are logged and maintained in order to detect suspicious activity. Upon detection of suspicious activities, user accounts should be temporarily suspended so that an audit can be conducted to confirm the suspicious activity to prevent a potential attack. |

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 | 04/09/2023 | Revised policies to implement DevSecOps for Security Policies, Principles, and Coding Standards. | Nicolas W. DeFrancisco |  |
| [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 |