**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 you validate input coming from any untrusted sources. Implementing proper input validation is crucial for minimizing software vulnerabilities. Exercise caution when dealing with external data sources, such as command line arguments, network interfaces, environmental variables, and user-controlled files. It's essential to be skeptical of the data received from these sources and perform thorough validation checks to mitigate potential security risks. |
| 1. Heed Compiler Warnings | When compiling your code, make sure to enable the highest warning level provided by your compiler. Pay attention to the warnings generated and modify your code accordingly to address them [C MSC00-A, C++ MSC00-A]. Additionally, utilize static and dynamic analysis tools to identify and fix any other potential security vulnerabilities in your code. |
| 1. Architect and Design for Security Policies | Designing software with a focus on security involves implementing and enforcing security policies. One approach is to create a software architecture that supports these policies. For instance, if your system needs different levels of privileges at different stages, you can divide it into separate intercommunicating subsystems, each with its own suitable set of privileges. By doing so, you establish boundaries and control access, ensuring that each subsystem operates with the appropriate level of security. |
| 1. Keep It Simple | Maintain a simple and concise design [Saltzer 74, Saltzer 75]. Complexity in design raises the risk of errors during implementation, configuration, and usage. Moreover, as security mechanisms become more intricate, the effort required to attain a suitable level of assurance significantly escalates. |
| 1. Default Deny | Ensure access decisions are based on permission rather than exclusion. By default, access should be denied, and the protection scheme should define the specific conditions under which access is allowed [Saltzer 74, Saltzer 75]. |
| 1. Adhere to the Principle of Least Privilege | Allow processes to operate with the minimum required privileges to accomplish their tasks. Elevated permissions should only be utilized for the shortest duration necessary to complete privileged operations. By adopting this practice, the likelihood of attackers being able to execute unauthorized code with elevated privileges is reduced [Saltzer 74, Saltzer 75]. |
| 1. Sanitize Data Sent to Other Systems | To ensure the security of complex subsystems, it is essential to sanitize all data before passing it to these subsystems. This applies particularly to command shells, relational databases, and commercial off-the-shelf (COTS) components. By doing so, you can mitigate the risk of attackers exploiting vulnerabilities through injection attacks such as SQL or command injection. |
| 1. Practice Defense in Depth | Employing multiple defensive measures is crucial for mitigating risks. By implementing a layered approach to security, vulnerabilities that may exist in one layer can be intercepted by subsequent layers, preventing them from being exploited or minimizing the impact of successful exploits. One way to achieve this is by combining secure programming practices with secure runtime environments. This combination reduces the chances of vulnerabilities present in the code at deployment from being exploited in the operational environment [Seacord 05]. |
| 1. Use Effective Quality Assurance Techniques | Utilizing robust quality assurance techniques can play a crucial role in detecting and mitigating vulnerabilities. Effective quality assurance programs should include practices such as fuzz testing, penetration testing, and source code audits. Incorporating independent security reviews can further enhance the security of systems. External reviewers offer an unbiased perspective, which can be valuable in uncovering and rectifying invalid assumptions [Seacord 05]. |
| 1. Adopt a Secure Coding Standard | Ensure the development process includes the establishment and implementation of a secure coding standard tailored to the specific programming language and platform being used. The coding standard should outline guidelines and best practices that focus on security considerations, aiming to prevent vulnerabilities and ensure the development of secure software. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | [Do not write syntactically ambiguous declarations](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL53-CPP.+Do+not+write+syntactically+ambiguous+declarations) |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | It is possible to devise syntax that can ambiguously be interpreted as either an expression statement or a declaration. Syntax of this sort is called a *vexing parse* because the compiler must use disambiguation rules to determine the semantic results. 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. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates a syntactic ambiguity issue where an anonymous local variable of type std::unique\_lock fails to lock and unlock the mutex due to the incorrect interpretation of the declaration. Consequently, the mutex object remains unlocked. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex>(m);  shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lock object is given an identifier (other than m) and the proper converting constructor is called. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex> lock(m);  shared\_resource += 42;  } |

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

| **Principles(s):** Heed Compiler Warnings, Keep It Simple  By adhering to the "Do not write syntactically ambiguous declarations" guideline, developers can reduce potential compiler warnings (following the "Heed Compiler Warnings" principle) and maintain simplicity in the code (following the "Keep It Simple" principle), which ultimately contributes to writing more secure and reliable code. |
| --- |

**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 |
| LDRA tool suite | 9.7.1 | **296 S** | Partially implemented |
| Parasoft C/C++ test | 2023.1 | **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. |

#### Coding Standard 2

| **Coding Standard** | **Label** | [Do not cast to an out-of-range enumeration value](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value) |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Enumerations in C++ come in two forms: scoped enumerations in which the underlying type is fixed and unscoped enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. 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. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example casts an integer value to an enumeration type and then attempts to check if it falls within the valid range of the enumeration. However, if the value is outside the valid range, it may lead to unspecified behavior due to the cast. |
| 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);  } |

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

| **Principles(s):** Validate Input Data, Keep It Simple  The coding standard emphasizes the importance of validating input data before casting to an enumeration value. This validation helps mitigate the risk of potential vulnerabilities that may occur due to operating on unspecified values. Additionally, the standard adheres to the principle of keeping it simple by implementing a straightforward check to ensure the arithmetic value falls within the enumeration's range before performing the cast operation, thereby minimizing the risk of errors during implementation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **cast-integer-to-enum** | Partially checked |
| CodeSonar | 7.3p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-INT50-a** | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| RuleChecker | 22.10 | **cast-integer-to-enum** | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | [Use valid references, pointers, and iterators to reference elements of a basic\_string](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR52-CPP.+Use+valid+references%2C+pointers%2C+and+iterators+to+reference+elements+of+a+basic_string) |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Since std::basic\_string is a container of characters, this rule is a specific instance of CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container. As a container, it supports iterators just like other containers in the Standard Template Library. However, the std::basic\_string template class has unusual invalidation semantics. Examples of standard library functions taking a reference to non-const std::basic\_string are std::swap(), ::operator>>(basic\_istream &, string &), and std::getline().  Do not use an invalidated reference, pointer, or iterator because doing so results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example copies input into a std::string, replacing semicolon (;) characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

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

| **Principles(s):** Validate Input Data, Sanitize Data Sent to Other Systems  The relevant coding standards advocate for using valid references, pointers, and iterators for basic\_string to prevent undefined behavior and implements data sanitization to protect against potential injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | **ALLOC.UAF** | Use After Free |
| Helix QAC | 2023.1 | **DF4746, DF4747, DF4748, DF4749** |  |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-STR52-a** | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2023a | **CERT C++: STR52-CPP** | Checks for use of invalid string iterator (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | [Do not access freed memory](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory) |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, 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. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, 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;  } |

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

| **Principles(s):** Validate Input Data, Use Effective Quality Assurance Techniques  The coding standard adheres to the principles of validating input data and using effective quality assurance techniques. It emphasizes not accessing freed memory through proper input validation and testing, which helps prevent the exploitation of dangling pointers and potential security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | **ALLOC.UAF** | Use after free |
| LDRA tool suite | 9.7.1 | **483 S, 484 S** | Partially implemented |
| Parasoft C/C++ test | 2023.1 | **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   Rule partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | [Detect and handle memory allocation errors](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors) |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example creates an array of int without checking the result of the allocation using ::operator new. Since the function is marked as noexcept, the caller assumes no exceptions will be thrown, but ::operator new can potentially throw an exception if the allocation fails, leading to abnormal program termination. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // Handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

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

| **Principles(s):** Validate Input Data, Heed Compiler Warnings  The coding standard adheres to the principles of "Validate Input Data" by ensuring proper handling of memory allocation errors and "Heed Compiler Warnings" by recommending the use of the nonthrowing form of the memory allocation operator and enabling the highest compiler warning level to catch potential issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | **CHECKED\_RETURN** | Finds inconsistencies in how function call return values are handled |
| LDRA tool suite | 9.7.1 | **45 D** | Partially implemented |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-MEM52-a**  **CERT\_CPP-MEM52-b** | Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Polyspace Bug Finder | R2023a | **CERT C++: MEM52-CPP** | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | [Do not subtract iterators that do not refer to the same container](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR54-CPP.+Do+not+subtract+iterators+that+do+not+refer+to+the+same+container) |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | When two pointers are subtracted, both must point to elements of the same array object or to one past the last element of the array object; the result is the difference of the subscripts of the two array elements. Similarly, when two iterators are subtracted (including via std::distance()), both iterators must refer to the same container object or must be obtained via a call to end() (or cend()) on the same container object. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to determine whether the pointer test is within the range [r, r + n]. However, when test does not point within the given range, as in this example, the subtraction produces undefined behavior. |
| #include <cstddef>  #include <iostream>    template <typename Ty>  bool in\_range(const Ty \*test, const Ty \*r, size\_t n) {  return 0 < (test - r) && (test - r) < (std::ptrdiff\_t)n;  }    void f() {  double foo[10];  double \*x = &foo[0];  double bar;  std::cout << std::boolalpha << in\_range(&bar, x, 10);  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates a fully portable, but likely inefficient, implementation of in\_range() that compares test against each possible address in the range [r, n]. A compliant solution that is both efficient and fully portable is currently unknown. |
| #include <iostream>    template <typename Ty>  bool in\_range(const Ty \*test, const Ty \*r, size\_t n) {  auto \*cur = reinterpret\_cast<const unsigned char \*>(r);  auto \*end = reinterpret\_cast<const unsigned char \*>(r + n);  auto \*testPtr = reinterpret\_cast<const unsigned char \*>(test);    for (; cur != end; ++cur) {  if (cur == testPtr) {  return true;  }  }  return false;  }    void f() {  double foo[10];  double \*x = &foo[0];  double bar;  std::cout << std::boolalpha << in\_range(&bar, x, 10);  } |

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

| **Principles(s):** Validate Input Data, Keep It Simple  The coding standard enforces input validation by disallowing the subtraction of iterators that do not refer to the same container or were not obtained via end() (or cend()) on the same container object. This simplifies the code design and minimizes potential vulnerabilities caused by invalid iterator manipulations and complexities in the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | **LANG.STRUCT.CUP**  **LANG.STRUCT.SUP** | Comparison of Unrelated Pointers  Subtraction of Unrelated Pointers |
| LDRA tool suite | 9.7.1 | **70 S, 87 S, 437 S, 438 S** | Enhanced Enforcement |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-CTR54-a** **CERT\_CPP-CTR54-b** **CERT\_CPP-CTR54-c** | Do not compare iterators from different containers Do not compare two unrelated pointers Do not subtract two pointers that do not address elements of the same array |
| Polyspace Bug Finder | R2023a | **CERT C++: CTR54-CPP** | Checks for subtraction or comparison between iterators from different containers (rule partially covered). |

#### Coding Standard 7

| **Coding Standard** | **Label** | [Handle all exceptions thrown before main() begins executing](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR58-CPP.+Handle+all+exceptions+thrown+before+main%28%29+begins+executing) |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When declaring an object with static or thread storage duration, and that object is not declared within a function block scope, the type's constructor must be declared noexcept and must comply with ERR55-CPP. Honor exception specifications. Additionally, the initializer for such a declaration, if any, must not throw an uncaught exception (including from any implicitly constructed objects that are created as a part of the initialization). If an uncaught exception is thrown before main() is executed, or if an uncaught exception is thrown after main() has finished executing, there are no further opportunities to handle the exception and it results in implementation-defined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the constructor for S may throw an exception that is not caught when globalS is constructed during program startup. |
| struct S {  S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| This solution involves changing the globalS variable into a local variable with static storage duration. This modification enables catching exceptions during object construction by deferring the execution of the constructor until the first invocation of the function globalS(). |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** Validate Input Data, Keep It Simple  The coding standard requires that objects with static or thread storage duration should have their constructors declared as noexcept. It also mandates that the initializer for such objects should not throw uncaught exceptions during initialization, as there would be no further opportunities to handle those exceptions before or after main() execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **potentially-throwing-static-initialization** | Partially checked |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-ERR58-a** | Exceptions shall be raised only after start-up and before termination of the program |
| Polyspace Bug Finder | R2023a | **CERT C++: ERR58-CPP** | Checks for exceptions raised during program startup (rule fully covered) |
| RuleChecker | 22.10 | **potentially-throwing-static-initialization** | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | [Range check element access](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR53-CPP.+Range+check+element+access) |
| --- | --- | --- |
| **String Correctedness** | [STD-008-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 uses the std::basic\_string::at() function, which behaves in a similar fashion to the index operator[] but throws a std::out\_of\_range exception if pos >= size(). |
| #include <stdexcept>  #include <string>  extern std::size\_t get\_index();    void f() {  std::string s("01234567");  try {  s.at(get\_index()) = '1';  } catch (std::out\_of\_range &) {  // Handle error  }  } |

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

| **Principles(s):** Validate Input Data, Keep It Simple  The coding standard applies the principles of validating input data to prevent undefined behavior when accessing elements outside the valid range of the std::string. Additionally, simplifying the approach by using range checks or error handling mechanisms would improve code clarity and reduce the risk of unexpected behavior during implementation and usage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| 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 |
| Astrée | 22.10 | **assert\_failure** |  |
| Parasoft C/C++ test | 2023.1 | **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 tainted offset   Rule partially covered. |

#### Coding Standard 9

| **Coding Standard** | **Label** | [Do not let exceptions escape from destructors or deallocation functions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL57-CPP.+Do+not+let+exceptions+escape+from+destructors+or+deallocation+functions) |
| --- | --- | --- |
| **Exceptions** | [STD-009-CPP] | Under certain circumstances, terminating a destructor, operator delete, or operator delete[] by throwing an exception can trigger undefined behavior. Deallocation functions (object, array, and placement forms at either global or class scope) must not terminate by throwing an exception. Do not declare such functions to be noexcept(false). However, it is acceptable to rely on the implicit noexcept(true) specification or declare noexcept explicitly on the function signature. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the class destructor does not meet the implicit noexcept guarantee because it may throw an exception even if it was called as the result of an exception being thrown. Consequently, it is declared as noexcept(false) but still can trigger undefined behavior. |
| #include <stdexcept>    class S {  bool has\_error() const;    public:  ~S() noexcept(false) {  // Normal processing  if (has\_error()) {  throw std::logic\_error("Something bad");  }  }  }; |

| **Compliant Code** |
| --- |
| A compliant destructor should either perform operations that do not throw exceptions or handle all exceptions within its scope without rethrowing them. The difference between a compliant solution and a noncompliant example is the explicit return statement in the destructor, preventing control from reaching the end of the exception handler and ensuring that the destructor does not terminate by throwing an exception. |
| class SomeClass {  Bad bad\_member;  public:  ~SomeClass()  try {  // ...  } catch(...) {  // Catch exceptions thrown from noncompliant destructors of  // member objects or base class subobjects.    // NOTE: Flowing off the end of a destructor function-try-block causes  // the caught exception to be implicitly rethrown, but an explicit  // return statement will prevent that from happening.  return;  }  }; |

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

| **Principles(s):** Validate Input Data, Keep It Simple  The coding standard ensures proper error handling by not allowing exceptions to escape from destructors or deallocation functions, aligning with the principle of "Validate Input Data." Additionally, it follows the principle of "Keep It Simple" by promoting a straightforward approach to error handling, reducing complexity and potential security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **destructor-without-noexcept** **delete-without-noexcept** | Fully checked |
| LDRA tool suite | 9.7.1 | **453 S** | Partially implemented |
| Polyspace Bug Finder | R2023a | **CERT C++: DCL57-CPP** | Checks for class destructors exiting with an exception (rule partially covered) |
| RuleChecker | 22.10 | **destructor-without-noexcept**  **delete-without-noexcept** | Fully checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | [Use offsetof() on valid types and members](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP59-CPP.+Use+offsetof%28%29+on+valid+types+and+members) |
| --- | --- | --- |
| **Data Types** | [STD-010-CPP] | The offsetof() macro is defined by the C Standard as a portable way to determine the offset, expressed in bytes, from the start of the object to a given member of that object. When specifying the type argument for the offsetof() macro, pass only a standard-layout class. The full description of a standard-layout class can be found in paragraph 7 of the [class] clause of the C++ Standard, or the type can be checked with the std::is\_standard\_layout<> type trait. When specifying the member designator argument for the offsetof() macro, do not pass a bit-field, static data member, or function member. Passing an invalid type or member to the offsetof() macro is undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a type that is not a standard-layout class is passed to the offsetof() macro, resulting in undefined behavior. |
| #include <cstddef>    struct D {  virtual void f() {}  int i;  };    void f() {  size\_t off = offsetof(D, i);  // ...  } |

| **Compliant Code** |
| --- |
| It is not possible to determine the offset to i within D because D is not a standard-layout class. However, it is possible to make a standard-layout class within D if this functionality is critical to the application, as demonstrated by this compliant solution. |
| #include <cstddef>    struct D {  virtual void f() {}  struct InnerStandardLayout {  int i;  } inner;  };    void f() {  size\_t off = offsetof(D::InnerStandardLayout, i);  // ...  } |

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

| **Principles(s):** Validate Input Data, Architect and Design for Security Policies  The coding standard "Use offsetof() on valid types and members" aligns with the principles of validating input data to minimize vulnerabilities and architecting for security policies by enforcing the usage of standard-layout classes, ensuring secure and well-defined behavior of the macro. By following this standard, developers establish boundaries, prevent potential security risks, and maintain a secure software design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Winvalid-offsetof | Emits an error diagnostic on invalid member designators, and emits a warning diagnostic on invalid types. |
| GCC | 4.9 | -Winvalid-offsetof | Emits an error diagnostic on invalid member designators, and emits a warning diagnostic on invalid types. |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-EXP59-a** | Use offsetof() on valid types and members |
| Polyspace Bug Finder | R2023a | **CERT C++: EXP59-CPP** | Checks use of offsetof macro with nonstandard layout class (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.

Automation will be extensively used throughout the DevSecOps lifecycle to enforce and comply with defined standards. Pre-production, tools like Coverity, Helix QAC, GCC, Clang, Parasoft C/C++ test, Polyspace Bug Finder, LDRA tool suite, Astrée, RuleChecker, and CodeSonar will aid in threat assessment, change analysis, and backlog prioritization. Design will embrace security test-driven practices and adhere to OWASP guidelines. In the Build phase, automation will ensure secure builds and trusted repository usage.

During the Verify and Test stage, automation will conduct vulnerability scans, validate trusted sources, and perform comprehensive testing. In production, automation will assist in configuring, deploying, and conducting security checks and penetration tests. It will also monitor logs, utilize SIEM, detect intrusions, and enable rapid response mechanisms. The goal is to maintain and stabilize the software by continually assessing security compliance and restoring stability after any incidents.

### 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 | Low | Unlikely | Medium | High | 3 |
| STD-002-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-003-CPP | High | Probable | High | Medium | 2 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-007-CPP | Low | Likely | Low | Low | 2 |
| STD-008-CPP | High | Unlikely | Medium | Medium | 2 |
| STD-009-CPP | Low | Likely | Medium | Medium | 2 |
| STD-010-CPP | Medium | Unlikely | Medium | Low | 3 |

### 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 | In encryption at rest, data protection is provided to stored data, typically in physical formats, such as on a disk. To prevent the mishandling of sensitive data if hardware is compromised, the most common implementation of this method is by encrypting hard drives with secure encryption keys. |
| Encryption at flight | In encryption at flight, data protection is provided to information that is moved across devices through a network or similar. Information that is handled through a network is especially vulnerable to security risks. One method to counter this is to implement a system where information is transferred via encrypted session keys, which are generated randomly at each session. |
| Encryption in use | In encryption in use, data protection is provided in any scenario where data is located on a non-persistent digital state, such as the CPU or RAM. The optimal method to protect against the unauthorized viewing of data is to implement full memory encryption, which is typically achieved by using a single encryption key to encrypt the entire physical memory of a system. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | To achieve a secure authentication process, user logins will utilize two-factor authentication. User information will be encrypted and stored in a safe location, where a verification against this database will take place upon login. |
| Authorization | For the authorization policy, we aim to maintain a strict limited access of sensitive data to specific users, which will be audited and reviewed whenever any changes to the user count are made. Each users’ access level will be determined based on their role within the team. |
| Accounting | For our accounting policy, we will be maintaining a database of system logs that determines system usage, changes to the database, and file access history per user. In addition, we will be dedicating resources towards monitoring these logs, and implementing security notifications as necessary. |

**\***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 | 07/21/2023 | Initial Research | Jacob Winters | Jacob Winters |
| 3.0 | 08/02/2023 | Finalized Document | Jacob Winters | Jacob Winters |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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