**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 | Data Validation is crucial in the development of a system. By ensuring data is validated, one can ensure the data’s integrity. All data sources must be considered when validating input. Input validation must be achieved to ensure data is safe for processing in the code or when communicating with other components. All input data must be treated as untrusted and validated properly. One example of this would be to ensure size arguments and length arrays are validated in a specific range. |
| 1. Heed Compiler Warnings | Create code in an iterative manner, compiling and testing code often. Warnings may allow the code to still run, but may be just that, a warning, for an area of the code that may be susceptible to vulnerabilities and attacks. Secure code by paying close attention to warnings and analyze them thoroughly. |
| 1. Architect and Design for Security Policies | Design code with security in mind throughout the development process. By utilizing professionally recognized secure coding standards, such as SEI Cert C++ coding standards, one can provide a secure product that still meets all user and customer requirements. |
| 1. Keep It Simple | When designing code, one should try to keep it simple in all aspects of the development. Keep components limited to single purpose as much as possible. By doing so, developers can keep security mechanisms simpler and will be able to ensure components are functioning correctly at time of compilations and implementation. |
| 1. Default Deny | By default, permissions and access to a system should be set as denied. One should base a decision in a system on permission rather than exclusion. All access to a system will be denied unless valid permission is granted once input is validated. |
| 1. Adhere to the Principle of Least Privilege | Access to a system will require certain privileges to certain validated data. Components in the system should function with the least amount of privileges possible and still maintain functionality. In accordance to this principle, higher privilege access should complete the desired task in the shortest time possible, limiting the time the access is granted, reducing time an attack can occur to the particular level of access. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data that is sent throughout the system Sanitize data to ensure it is free of vulnerabilities or sensitive information before it is sent to another system. Failure to sanitize data properly may lead to unsecure data being sent with the possibility of exploitation. |
| 1. Practice Defense in Depth | Practice DiD by adding layers of security to the system. Evaluate the appropriate security needed for the particular system. Define layers of security being used and the purpose of the layers. Ensure layers implemented do not align with other layers to avoid vulnerabilities being disregarded due to lack of diversity in the layers. |
| 1. Use Effective Quality Assurance Techniques | Use proper quality assurance standards in implementation. QA can help find vulnerabilities in the development stage before implementation of the system. Use of proper testing methods and techniques can help in this. Test Automation can reduce the time spent manually testing the system. When possible, implement an external security/ test review to allow for alternate perspectives and tests to be implemented on the system. |
| 1. Adopt a Secure Coding Standard | One should adhere to a secure coding standard they adopt in the development of a system by doing so, one will keep security at the forefront of the development, making it much easier to implement other policies throughout. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not define a C-style variadic function. A variadic function using a C-style variadic function has no mechanisms to check the type safety of arguments being passed to the function or to check that the number of arguments being passed matches the semantics of the function definition. This can lead to an undefined behavior |

| **Noncompliant Code** |
| --- |
|  |
| The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. By passing any other type than int will result in undefined behavior as well |
| #include <cstdarg>    int add(int first, int second, ...) {  int r = first + second;  va\_list va;  va\_start(va, second);  while (int v = va\_arg(va, int)) {  r += v;  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. The use of std::enable\_if to ensure that any nonintegral argument value results in an ill-formed program |
| #include <type\_traits>    **template** <**typename** Arg, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Arg s) { **return** f + s; }    **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Ts... rest) {  **return** f + add(rest...)} |

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

| **Principles(s):**   * 1.Validating Input Data – ensuring proper inputs * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 20.10 | function-ellipsis | Fully checked |
| Parasoft C/C++test | 2022.1 | **CERT\_CPP-DCL50-a** | Functions shall not be defined with a variable number of arguments |
| Polyspace Bug Finder | R2022b | CERT C++: DCL50-CPP | Checks for function definition with ellipsis notation (rule fully covered) |
| RuleChecker | 20.10 | function-ellipsis | Fully checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not declare or define a reserved identifier. Defining or declaring identifiers with reserved names may lead to undefined behavior. Reserved words and the names of C standard library functions should not be used as identifiers in the code |

| **Noncompliant Code** |
| --- |
|  |
| Header guards are trailed by an underscore, leading to potential undefined behavior. Such a name may clash with reserved names defined by the implementation of the C++ standard template library in its headers or with reserved names implicitly predefined by the compiler even when no C++ standard library header is included. |
| #ifndef \_MY\_HEADER\_H\_  #define \_MY\_HEADER\_H\_    // Contents of <my\_header.h>  #endif // \_MY\_HEADER\_H\_ |

| **Compliant Code** |
| --- |
| In the compliant code, the header guards are not trailed by underscores |
| #ifndef MY\_HEADER\_H  #define MY\_HEADER\_H    // Contents of <my\_header.h>    #endif // MY\_HEADER\_H |

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

| **Principles(s):**   * 2.Heed compiler warnings – always pay attention to warnings as they are there for a reason * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | reserved-identifier | Partially checked |
| Clang | 3.9 | -Wreserved-id-macro  -Wuser-defined-literals | The -Wreserved-id-macro flag is not enabled by default or with -Wall, but is enabled with -Weverything. This flag does not  catch all instances of this rule, such as redefining reserved names. |
| CodeSonar | 7.1p0 | LANG.ID.NU.MK  LANG.STRUCT.DECL.RESERVED | Macro name is C keyword  Declaration of reserved name |
| LDRA tool suite | 9.7.1 | 86 S, 218 S, 219 S, 580 S | Fully implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL51-a  CERT\_CPP-DCL51-b  CERT\_CPP-DCL51-c  CERT\_CPP-DCL51-d  CERT\_CPP-DCL51-e  CERT\_CPP-DCL51-f | Do not #define or #undef identifiers with names which start with underscore  Do not redefine reserved words  Do not #define nor #undef identifier 'defined'  The names of standard library macros, objects and functions shall not be reused  The names of standard library macros, objects and functions shall not be reused (C90)  The names of standard library macros, objects and functions shall not be reused (C99) |
| Polyspace Bug Finder | R2022b | CERT C++: DCL51-CPP | Checks for redefinitions of reserved identifiers (rule partially covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Never qualify a reference type with const or volatile |

| **Noncompliant Code** |
| --- |
| This code is ill formed, although p is correctly declared. The const qualifier of p leads to an ill formed declaration and cannot be modified. |
| #include <iostream>    void f(char c) {  const char &p = c;  p = 'p'; // Error: read-only variable is not assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| By removing the const qualifier the declaration of p is correctly formed, allowing the code to not store undefined data incorrectly. |
| #include <iostream>    void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):**   * 2.Heed compiler warnings – always pay attention to warnings as they are there for a reason * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | CERT\_CPP-DCL52-a | |  | | --- | | Never qualify a reference type with  'const' or 'volatile' | |
| Polyspace Bug Finder | R2022b | CERT C++: DCL52-CPP | Checks for:  const-qualified reference types  Modification of const-qualified reference types  Rule fully covered. |
| Clang | 3.9 |  | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options |
| SonarQube C/C++ Plugin | 4.10 | S3708 |  |
| Klocwork | 2022.3 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE |  |
| Helix QAC | 2022.3 | C++0014 |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not write syntactically ambiguous declarations |

| **Noncompliant Code** |
| --- |
| Declaration is syntactically ambiguous of std::unique\_lock, as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named m and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked |
| #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** |
| --- |
| The compliant code gives m a proper identifier, making it not ambiguous. A proper converting constructor is then called as well. |
| #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):**   * 1.Validating Input Data – ensuring proper inputs * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |
| LDRA tool suite | 9.7.1 | 296 S | Partially implemented |
| Parasoft C/C++test | 2022.1 | 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 | R2022b | 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. |
| PRQA QA-C++ | 4.4 | 2502, 2510 |  |
| SonarQube C/C++ Plugin | 4.10 | S3468 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Overload allocation and deallocation functions as a pair in the same scope. If an allocation function is overloaded in a particular scope, the corresponding deallocation function must also be overloaded in the same scope. |

| **Noncompliant Code** |
| --- |
| Allocation is declared in a global scope. The lack of a deallocation function in this same scope can lead to undefined behavior. In attempting to delete a function that was overloaded with the allocation function, undefined behavior would be present. |
| #include <Windows.h>  #include <new>    void \*operator new(std::size\_t size) noexcept(false) {  static HANDLE h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }    // No corresponding global delete operator defined. |

| **Compliant Code** |
| --- |
| This example shows the proper use of deallocation in the same scope as the allocation. Both allocation and deallocation are called on the same scope public. |
| #include <Windows.h>  #include <new>    class HeapAllocator {  static HANDLE h;  static bool init;    public:  static void \*alloc(std::size\_t size) noexcept(false) {  if (!init) {  h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  init = true;  }    if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }    static void dealloc(void \*ptr) noexcept {  if (h) {  (void)::HeapFree(h, 0, ptr);  }  }  };    HANDLE HeapAllocator::h = nullptr;  bool HeapAllocator::init = false;    void \*operator new(std::size\_t size) noexcept(false) {  return HeapAllocator::alloc(size);  }    void operator delete(void \*ptr) noexcept {  return HeapAllocator::dealloc(ptr);  } |

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

| **Principles(s):**   * 1.Validating Input Data – ensuring proper inputs * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | new-delete-pairwise | Partially checked |
| Clang | 3.9 | misc-new-delete-overloads | Checked with clang-tidy. |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL54-a | Always provide new and delete together |
| Polyspace Bug Finder | R2022b | CERT C++: DCL54-CPP | Checks for mismatch between overloaded operator new and operator delete (rule fully covered) |
| RuleChecker | 20.10 | new-delete-pairwise | Partially checked |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Avoid information leakage when passing a class object across a trust boundary. The data passing needs to be verified before it can cause issues |

| **Noncompliant Code** |
| --- |
| The code copies data from arg to the user space while running in kernel space. Padding the bits outside of the object may lead to bit padding containing sensitive data that may get leaked when data is copied to the user space. |
| #include <cstddef>    struct test {  int a;  char b;  int c;  };    // Safely copy bytes to user space  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {  test arg{1, 2, 3};  copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |

| **Compliant Code** |
| --- |
| This code solution serializes the structure data before copying it to an untrusted context. This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the copy\_to\_user() function would need to unpack it to recreate the original, padded structure |
| #include <cstddef>    struct test {  int a;  char b;  int c;  };    // Safely copy bytes to user space  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {  test arg{};    arg.a = 1;  arg.b = 2;  arg.c = 3;    copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |

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

| **Principles(s):**   * 1.Validating Input Data – ensuring proper inputs * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 7.Sanitize data sent to other systems – keeping data sent to other systems as only required/authorized * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | MISC.PADDING.POTB | Padding Passed Across a Trust Boundary |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL55-a | A pointer to a structure should not be passed to a function that can copy data to the user space |
| Polyspace Bug Finder | R2022b | CERT C++: DCL55-CPP | Checks for information leakage due to structure padding (rule partially covered) |
| Helix QAC | 2022.3 | C++4941, C++4942, C++4943 |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Avoid cycles during initialization of static objects. If a function is reentered during the constant initialization of a static object inside that function, the behavior of the program is undefined. |

| **Noncompliant Code** |
| --- |
| By using caching, the non-compliant code attempts to implement an efficient factorial function. Because the initialization of the static local array cache involves recursion, the behavior of the function is undefined, even though the recursion is not infinite. |
| #include <stdexcept>    int fact(int i) noexcept(false) {  if (i < 0) {  // Negative factorials are undefined.  throw std::domain\_error("i must be >= 0");  }    static const int cache[] = {  fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),  fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),  fact(12), fact(13), fact(14), fact(15), fact(16)  };    if (i < (sizeof(cache) / sizeof(int))) {  return cache[i];  }    return i > 0 ? i \* fact(i - 1) : 1;  } |

| **Compliant Code** |
| --- |
| The solution avoids initializing the static local array cache and instead relies on zero-initialization to determine whether each member of the array has been assigned a value yet. If no value is assigned yet, recursively computes its value. It then returns the cached value when possible or computes the value as needed. |
| #include <stdexcept>    int fact(int i) noexcept(false) {  if (i < 0) {  // Negative factorials are undefined.  throw std::domain\_error("i must be >= 0");  }    // Use the lazy-initialized cache.  static int cache[17];  if (i < (sizeof(cache) / sizeof(int))) {  if (0 == cache[i]) {  cache[i] = i > 0 ? i \* fact(i - 1) : 1;  }  return cache[i];  }    return i > 0 ? i \* fact(i - 1) : 1;  } |

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

| **Principles(s):**   * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.STRUCT.INIT.CYCLE  LANG.STRUCT.INIT.UNORDERED | Initialization Cycle  Unordered Initialization |
| LDRA tool suite | 9.7.1 | 6 D | Enhanced Enforcement |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL56-a | Avoid initialization order problems across translation units by replacing non-local static objects with local static objects |
| Polyspace Bug Finder | R2022b | CERT C++: DCL56-CPP | Checks for:  Recursive initialization of static variables  Undetermined initialization order of global variables  Rule fully covered. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | Do not let exceptions escape from destructors or deallocation functions. Under certain circumstances, terminating a destructor, operator delete, or operator delete[] by throwing an exception can trigger undefined behavior. |

| **Noncompliant Code** |
| --- |
| A global deallocation is declared noexcept(false) and throws an exception if some conditions are not properly met. However, throwing from a deallocation function results in undefined behavior. |
| #include <stdexcept>    bool perform\_dealloc(void \*);    void operator delete(void \*ptr) noexcept(false) {  if (perform\_dealloc(ptr)) {  throw std::logic\_error("Something bad");  }  } |

| **Compliant Code** |
| --- |
| The compliant solution does not throw exceptions in the event the deallocation fails but instead fails as gracefully as possible. |
| #include <cstdlib>  #include <stdexcept>    bool perform\_dealloc(void \*);  void log\_failure(const char \*);    void operator delete(void \*ptr) noexcept(true) {  if (perform\_dealloc(ptr)) {  log\_failure("Deallocation of pointer failed");  std::exit(1); // Fail, but still call destructors  }  } |

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

| **Principles(s):**   * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | destructor-without-noexcept  delete-without-noexcept | Fully checked |
| CodeSonar | 7.1p0 | LANG.STRUCT.EXCP.CATCH  LANG.STRUCT.EXCP.THROW | Use of catch  Use of throw |
| LDRA tool suite | 9.7.1 | 453 S | Partially implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL57-a  CERT\_CPP-DCL57-b | Never allow an exception to be thrown from a destructor, deallocation, and swap  Always catch exceptions |
| Polyspace Bug Finder | R2022b | CERT C++: DCL57-CPP | Checks for class destructors exiting with an exception (rule partially covered) |
| RuleChecker | 20.10 | destructor-without-noexcept  delete-without-noexcept | Fully checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | Do not modify the standard namespaces. Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions. One feature of namespaces is that they can be further extended, even within separate translation units. For instance, the following declarations are well-formed. |

| **Noncompliant Code** |
| --- |
| the declaration of x is added to the namespace std, resulting in undefined behavior. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| It assumes the programmer’s intention is to place x declaration into a namespace to prevent collisions with global identifiers. X is placed into a namespace without a reserved name |
| namespace nonstd {  int x;  } |

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

| **Principles(s):**   * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice * 9.Use Effective Quality Assurance Techniques – making tests that are as effective as possible * 10.Adopt a secure coding standard – making security a priority helps prevent vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.STRUCT.DECL.SNM | Modification of Standard Namespaces |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces 'std' and 'posix' |
| Polyspace Bug Finder | R2022b | CERT C++: DCL58-CPP | Checks for modification of standard namespaces (rule fully covered) |
| PRQA QA-C++ | 4.4 | 4032, 4035, 4631 |  |
| SonarQube C/C++ Plugin | 4.10 | S3470 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | Do not define an unnamed namespace in a header file. Unnamed namespaces are used to define a namespace that is unique to the translation unit, where the names contained within have internal linkage by default. |

| **Noncompliant Code** |
| --- |
| variable v is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of v and then assigns a new value into it. By not having a defined namespace and v if defined within it, each translation unit operates on its own instance of v, resulting in unexpected output.. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    namespace {  int v;  }    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  #include <iostream>    void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }    // b.cpp  #include "a.h"  #include <iostream>    void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }    int main() {  extern void f();  f(); // Prints v, sets it to 42  g(); // Prints v, sets it to 100  f();  g();  } |

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

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

| **Principles(s):**   * 3.Architect and Design for Security Policies – building code to prevent vulnerabilities * 4.Keep it simple – always applies as keeping code as lightweight as possible is best practice |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | unnamed-namespace-header | Fully checked |
| Clang | 3.9 | cert-dcl59-cpp | Checked by clang-tidy |
| CodeSonar | 7.1p0 | LANG.STRUCT.DECL.ANH | Anonymous Namespace in Header File |
| LDRA tool suite | 9.7.1 | 286 S, 512 S | Fully implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL59-a | There shall be no unnamed namespaces in header files |
| Polyspace Bug Finder | R2022b | CERT C++: DCL59-CPP | Checks for unnamed namespaces in header files (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 of testing procedures will be implemented throughout the new practices of the DevSecOps diagram. As portions of a program will be developed, automated tests will be implemented throughout the iterations being developed. This can be triggered through new saves to a repository, whether it is on a physical machine or virtually through a cloud-based network the repository is linked to. This coverage over a network linked to the repository will allow for all parties involved in the development, design and testing to have the proper automation testing processes ran on their portions as well.

By iterating through a development process and adopting automation tests in this process, it will help catch vulnerabilities in the early stages of their creation, reducing the costs of potential fixes later in the development process. Identifying vulnerabilities in the early stages of creation can also point to potential vulnerabilities in the future. It essentially creates a roadmap for all parties to teach, train, and develop secure software by implementing past issues.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Probable | Medium | P12 | L1 |
| STD-002-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-003-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-004-CPP | Low | Unlikely | Medium | P3 | L2 |
| STD-005-CPP | Low | Probable | Low | P6 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-008-CPP | Low | Likely | Medium | P6 | L2 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-010-CPP | Medium | Unlikely | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest explains how data is stored and held on servers or in repositories. Examples of how we will implement this would be file-level encryption, database encryption, full disk encryption and Protection with Digital rights management. Defense in Depth will implement this as a final defense against potential data breaches. Even if all levels of a defense system are accessed by a hacker, the data they are attempting to access can still be encrypted and not looked at as viable. This will allow for protection of data from in-house unauthorized access due to vulnerabilities intentionally or unintentionally. |
| Encryption at flight | Data in flight is the description of data being transitioned from one system to another. In order to provide adequate security to data in the transfer stage, the use of encryption will be necessary. This encryption will be done with the use of a private key used to encrypt data from the sender and can only be accessed fully with the user’s public key on the other end. The authentication of the receiver of the data. If the data is intercepted by an unauthorized user, without the public key authentication and authorization, the data will be rendered as not viable. |
| Encryption in use | Encryption in use is refers to when data is directly in use. This policy is enforced by Identity and roll-based management. This uses the principle of Least Privilege. With least privilege being established, data that falls outside of the rolls / authorization levels. Will not be accessible to anyone and will remain encrypted to all parties not authorized to decrypt it. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The verification of a user’s credentials, no matter if they are authorized or not. The use of usernames, passwords, ids, pin, or security tokens can be used to authenticate a user. |
| Authorization | After authentication, authorization to the network enables the user to have specific permissions to areas through authorization. Principle of Least Privilege is used to allow the minimum access to users of certain levels while limiting authorization to other levels. |
| Accounting | The means of monitoring and capturing the events done by the user while accessing the network resources. It even monitors how long the user has access to the network. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.11 | 09/17/2022 | Milestone 1 Revision | Aaron Shipley | Aaron Shipley |
| 1.2 | 10/05/2022 | Project 1 Revision | Aaron Shipley | Aaron Shipley |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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