**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 | This principle emphasizes the importance of checking and validating all input data to ensure that only appropriate and safe data is processed by the system. This helps prevent the introduction of malicious data, such as through SQL injection attack. |
| 1. Heed Compiler Warnings | Compiler warnings alert developers to potential issues or errors in the code. Even if the code compiles successfully, these warnings may indicate vulnerabilities or bugs that could lead to security risks. Addressing these warnings helps maintain code quality and security. |
| 1. Architect and Design for Security Policies | Security is an important part of the software architecture and design. This involves considering security policies during the design phase, such as dividing the system into subsystems with varying levels of authorization and privilege. |
| 1. Keep It Simple | A simple design reduces the likelihood of errors and vulnerabilities in the code. Complexity can introduce security risks, so keeping the system and its security measures straightforward helps ensure a more secure and maintainable system. |
| 1. Default Deny | This principle means that access to resources should be denied by default. Only explicitly authorized users or processes are granted access, based on a well-defined security policy. This approach helps minimize unauthorized access and potential security breaches. |
| 1. Adhere to the Principle of Least Privilege | Systems and processes should operate with the minimum level of privilege necessary to perform their functions. This reduces the risk of an attacker gaining elevated privileges and limits the potential damage if a component is compromised. |
| 1. Sanitize Data Sent to Other Systems | It is quite important that before passing data to other systems or components, it should be sanitized to remove or neutralize potentially harmful content. This helps prevent attacks such as SQL injection and ensures that data exchanged between systems is safe and secure. |
| 1. Practice Defense in Depth | This principle involves implementing multiple layers of security measures to protect a system. If one layer is breached, additional layers can still provide protection. This approach increases the overall security of the system by providing multiple lines of defense against attacks. |
| 1. Use Effective Quality Assurance Techniques | Implementing strong testing methodologies such as penetration testing is important for identifying and addressing security vulnerabilities. Regular code audits and security reviews, both internal and external, help ensure that security issues are detected and resolved. |
| 1. Adopt a Secure Coding Standard | Following secure coding standards helps ensure that code is written with security in mind from the outset. These standards provide guidelines and best practices for avoiding common vulnerabilities and ensuring that the codebase remains secure throughout its lifecycle. |

### 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 cast to an out-of-range enumeration value |

| **Noncompliant Code** |
| --- |
| Verifies if a given value falls within the acceptable range of enumeration values. Casting the type may result in an inability to accurately represent the specified integer value. |
| enum EnumType {  First,  Second,  Third  }; void f(int intVar) {  EnumType enumVar = static\_cast  if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| The compliant approach verifies the value represented by the enumeration type before performing the conversion to ensure that the result is a valid, specified value. This process restricts the converted value to one specific enumerator type. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  // Check if intVar is within the range of valid enumeration values  if (intVar < First || intVar > Third) {  // Handle error: intVar is out of the valid range for EnumType  // For example, you might throw an exception or log an error message  // throw std::out\_of\_range("Invalid enumeration value");  return; // Exit the function or handle the error appropriately  }  // Safe to cast intVar to EnumType now, as it's within the valid range  EnumType enumVar = static\_cast<EnumType>(intVar);  // Continue with the rest of the function logic using enumVar  } |

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

| **Principles(s):** Validate Input Data -this principle is paramount in ensuring that mistakes are caught from the onset. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS – Studio | 7.07 | V1016 |  |
| Axivion Bauhaus Suite | 6.9.0 | CertC++ - INT50 |  |
| Helix QAC | 2024.2 | C++3162, C++3163, C++3164, C++3165 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002- CPP | Use valid references, pointers, and iterators to reference elements of a container |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the variable pos becomes invalid after the first call to insert(), leading to undefined behavior in subsequent loop iterations. |
| #include <vector>  void f(const int \*items, std::size\_t count) {  std::vector<int> v;  auto pos = v.begin();  for (std::size\_t i = 0; i < count; ++i, ++pos) {  v.push\_back(items[i] \* 2);  // Invalidates pos, as push\_back may cause reallocation  // and invalidation of iterators.  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, pos is reassigned to the iterator returned by d.insert, which ensures it remains valid with each insertion. This approach prevents undefined behavior by updating the iterator correctly after every insertion. |
| #include <deque>  void f(const double \*items, std::size\_t count) {  std::deque<double> d;  auto pos = d.begin();  for (std::size\_t i = 0; i < count; ++i) {  pos = d.insert(pos, items[i] + 41.0);  ++pos; // Advance pos after insertion to avoid invalidation  }  } |

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

| **Principles(s):**  Validate Input Data -by validating input data, we prevent out-of-bounds memory errors, which can occur when invalid iterators, references, or pointers are used.  Heed Compiler Warnings -compiler warnings can catch issues such as invalid iterator usage before they reach production, ensuring safer code.  Architect for Security and Design -educating the team on correct type usage and architectural considerations can prevent issues related to invalid references, iterators, and pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Overflow\_unpon\_dereference |  |
| Helix QAC | 2024.2 |  |  |
| Parasoft C/C++  test | 2023.06 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| PVS – Studio | 7.07 | V783 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003- CPP | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| In the following noncompliant code, a std::string is constructed directly from the result of std::getenv(). Since std::getenv() returns a null pointer if the environment variable does not exist or an error occurs, this can lead to undefined behavior. The code does not check if std::getenv() returned a valid pointer before using it to initialize the std::string. |
| #include <cstdlib>  #include <string>  void f() {  std::string tmp(std::getenv("TMP")); // Potentially undefined behavior if std::getenv() returns null  if (!tmp.empty()) {  // Handle the case where the environment variable exists  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the result of std::getenv() is first checked for null. If it is not null, it is used to initialize the std::string. If it is null, an empty string is used instead. This approach prevents undefined behavior by ensuring that the std::string is constructed with a valid value. |
| #include <cstdlib>  #include <string>  void f() {  const char \*tmpPtrVal = std::getenv("TMP"); // Check if getenv returns a valid pointer  std::string tmp(tmpPtrVal ? tmpPtrVal : ""); // Safely construct string with a default value if nullptr  if (!tmp.empty()) {  // Handle the case where the environment variable exists  // ...  }  } |

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

| **Principles(s):**  Avoiding Null Pointer Dereference -this principle emphasizes the importance of ensuring that a program does not attempt to dereference a null pointer, as doing so leads to undefined behavior. The consequences can range from abnormal program termination to, in more severe cases, the execution of arbitrary code, depending on the platform. The severity of this issue varies, being more critical in environments where such dereferencing can be exploited to execute arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 | C++3162, C++3163, C++3164, C++3165 | Identifying potential instances where a null pointer might be dereferenced, which could lead to undefined behavior or security vulnerabilities. |
| ParasoftC/C++ test | 2020.2 | CERT\_CPP-STR51-a | This checker is designed to automatically detect and prevent situations where a null pointer might be dereferenced, which aligns with the principle of avoiding undefined behavior and potential security risks associated with null pointer dereferencing in C++ code. |
| Astree | 20.10 | Assert\_failure | This checker helps ensure that the code adheres to safe practices by verifying that pointers are valid before dereferencing, thereby preventing undefined behavior or program crashes due to null pointer dereferencing. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004- CPP | Do not store already-owned pointer value in an unrelated smart pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, two std::shared\_ptr objects are initialized with the same raw pointer. When the local variable p2 is destroyed, it deletes the managed pointer. However, when the local variable p1 is subsequently destroyed, it tries to delete the same pointer again, leading to a double-free vulnerability. |
| #include <memory>  void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i); // p1 owns the pointer  std::shared\_ptr<int> p2(i); // p2 also owns the same pointer  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::shared\_ptr objects are related through copy construction, ensuring proper reference counting and avoiding double-free issues. The use of std::make\_shared() also enhances safety by eliminating the need to manually allocate a raw pointer. |
| #include <memory>  void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):**  Proper Memory Management and Deallocation -this principle emphasizes the importance of ensuring that pointers passed to deallocation functions, such as free() in C or delete in C++, were originally obtained from the corresponding allocation functions. Failing to adhere to this can result in undefined behavior, potentially leading to exploitable vulnerabilities such as double free, use-after-free, or memory corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Dangling\_pointer\_use | Detects the use of dangling pointers, which can lead to undefined behavior and security vulnerabilities. |
| Helix QAC | 2024.2 | C++3640 | Ensures that pointers passed to deallocation functions were obtained from the corresponding allocation functions. |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-MEM56-a | Ensures proper memory management by preventing issues like double free or invalid free. |
| PVS-Studio | 7.21 | V1006 | Detects potential issues related to incorrect usage of pointers, particularly in smart pointers. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005- CPP | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| In this example, the local variable space is used with the placement new operator to construct an object of type S. The resulting pointer is then incorrectly passed to ::operator delete(), leading to undefined behavior because ::operator delete() attempts to free memory not allocated by ::operator new(). |
| #include <iostream>  struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };  void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;  // ...  delete s1;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to ::operator delete() is removed, and s1's destructor is explicitly called. This is one of the rare cases where directly invoking a destructor is appropriate. |
| #include <iostream>  struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };  void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;  // ...  s1->~S(); // Explicitly call the destructor  } |

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

| **Principles(s):**  Proper Memory Management and Deallocation -this principle emphasizes that a pointer should only be passed to a deallocation function if it was originally allocated by the corresponding allocation function. Failing to adhere to this principle can result in undefined behavior, such as memory corruption or vulnerabilities, because the deallocation function may not correctly handle the pointer, leading to potential security risks or program crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Invalid\_dynamic\_memory\_allocation\_dangling\_pointer\_use | Dynamic memory management is improperly handled, specifically involving the use of pointers that have been deallocated or are otherwise invalid. |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free.  Always provide empty brackets [] for delete when deallocating arrays.  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor.  Properly deallocate dynamically allocated resources. |
| CodeSonar | 6.0p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM | Free non-heap variable  Double free  Type mismatch |
| SonarQube C/C++ Plugin | 4.10 | S1232 | This checker identifies potential problems with dangling pointers and invalid deallocation practices. The checker helps ensure that memory management operations are performed correctly, avoiding common pitfalls such as deallocating memory that was not dynamically allocated or using pointers that may have already been freed. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006- CPP | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| In this example, the assertion checks that the size of the timer structure is equal to the sum of the sizes of its members. However, this check is performed using assert(), which may be disabled in production builds by defining NDEBUG. Therefore, relying on assert() for essential runtime checks is unsafe, as it could lead to unexpected behavior if the assumption is violated and the assertion is disabled. |
| #include <assert.h>  struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a preprocessor conditional statement is used to ensure that the timer structure does not contain padding, which is crucial for the correct behavior of the code. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** Adopt a Secure Coding Standard -this practice is integral to maintaining a secure coding standard, as it ensures that certain conditions are checked during compilation, reducing the risk of runtime errors that could lead to security breaches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion  Bauhaus Suite | 6.9.0 | CERTC-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| CodeSonar | 6.0p0 | (customization) | Users have the option to create a custom check that identifies instances where the assert() macro is used. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007- CPP | Handle all exceptions thrown before main() begins executing |

| **Noncompliant Code** |
| --- |
| In this example, the constructor for S is marked as potentially throwing an exception (noexcept(false)). However, this exception is not handled when globalS is constructed during program startup. If an exception is thrown, it could lead to unexpected behavior or terminate the program. |
| struct S {  S() noexcept(false);  };  static S globalS; |

| **Compliant Code** |
| --- |
| This solution converts globalS into a local variable with static storage duration within a function. This ensures that any exceptions thrown during the construction of S can be caught, as the constructor will only be called the first time the function globalS() is invoked, rather than at program startup. This approach requires modifying the code to replace previous uses of globalS with a function call to 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):**  Adhere to the Principle of Least Privilege -exceptions that are not properly managed can cause a program to terminate unexpectedly. If such unhandled exceptions lead to a crash or abnormal termination, they can create opportunities for denial-of-service (DoS) attacks, where an attacker deliberately causes disruptions or crashes to make the system unavailable to legitimate users. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-eer58-cpp | Checked by clang-tidy |
| Rule Checker | 20.10 | potentially-throwing-static-  initialization | Partially checked |
| Parasoft C/C++  test | 2020.2 | CERT\_CPP-ERR58-a | Exceptions should be triggered only after the program has started and before it terminates. |
| Axivion Bauhaus Suite | 6.9.0 | CERTC++-ERR58 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input output | STD-008- CPP | Do not alternately input and output from a file stream without an intervening positioning call. |

| **Noncompliant Code** |
| --- |
| In this example, data is appended to a file and then read from the same file. However, since there is no repositioning of the file stream between the output and input operations, the behavior is undefined. |
| #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  file << "Output some data";  std::string str;  file >> str;  } |

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

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

| **Principles(s):**  Sanitize Data Sent to Other Systems -this principle emphasizes the importance of managing data interactions, such as input and output operations, to prevent undefined behavior and potential issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 |  |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP\_FIO50-a | Avoid alternating between input and output operations on a stream without performing a flush or positioning call in between. |
| Polyspace Bug Finder | R2024a | ECRT C++: FIO50-CPP | Checks for alternating between input and output operations on a stream without performing a flush or positioning call (rule fully addressed). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programing | STD-009- CPP | Do not invoke virtual functions from constructors or destructors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the base class B calls virtual functions seize() and release() within its constructor and destructor. This can lead to undefined behavior because the derived class D's implementation of these functions might not be fully initialized or properly destructed at these points. |
| struct B {  B() { seize(); }  virtual ~B() { release(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  B::seize(); // Call base class version  // Get derived resources...  }  void release() override {  // Release derived resources...  B::release(); // Call base class version  }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the base class B and derived class D use nonvirtual, private member functions for resource management. This avoids calling virtual functions from constructors and destructors, which can lead to undefined behavior. |
| class B {  private:  void seize\_mine();  void release\_mine();  public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }  protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };  class D : public B {  private:  void seize\_mine();  void release\_mine();  public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }  protected:  void seize() override {  B::seize();  seize\_mine();  }  void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):**  Adhere to the Principle of Least Privilege -by avoiding the invocation of virtual functions from constructors and destructors, you prevent the potential for undefined behavior and maintain proper initialization and destruction sequences. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P18 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-  alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Parasoft C/C++  test | 2023.1 | CERT\_CPP-OOP50-a  CERT\_CPP-OOP50-b  CERT\_CPP-OOP50-c  CERT\_CPP-OOP50-d | Refrain from calling virtual functions within constructors.  Refrain from calling virtual functions within destructors.  Avoid invoking a class's virtual functions from any of its constructors.  Avoid invoking a class's virtual functions from its destructor. |
| PVS-Studio | 20.10 | Virtual-call-in-customer | Fully checked |
| Helix QAC | 2024.2 |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | STD-010- CPP | Value returning functions must return a value from all exit paths. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function absolute\_value fails to return a value for non-negative inputs, leading to undefined behavior when the input is zero or positive. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  // Missing return statement for non-negative values  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function absolute\_value ensures that all code paths return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques -emphasizes the importance of ensuring that all code paths in functions that are supposed to return a value actually do so. Inadequate return statements can lead to undefined behavior and potential security vulnerabilities, affecting data integrity and system stability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++  test | 2023.1 | CERT\_CPP-MSC52-a | Every exit path from a function that has a non-void return type must include an explicit return statement with a value. |
| Polyspace Bug  Finder | R2024a | Cert C++: MSC52-a | Identifies instances where return statements are missing (rule partially addressed). |
| SonarQube  C/C++ Plugin | 9.9.0 | S1699 | Avoids potential class casting issues and enhance type safety |
| RuleChecker | 20.10 | Return-implicit | Fully checked |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Likely | High | P2 | 2 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | Low | Likely | Medium | P6 | L2 |
| STD-009-CPP | High | Medium | Medium | P18 | L2 |
| STD-010-CPP | Medium | Probable | Medium | P8 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | -this refers to the practice of encrypting data that is stored on disk or other storage media to protect it from unauthorized access. It is useful in the implementation of full-disk encryption at the server level to protect all stored data, including files and databases. It protects data from unauthorized access in case of physical theft or unauthorized access to storage media. Ensures data confidentiality and complies with data protection regulations by keeping sensitive information secure. |
| Encryption in flight | -involves securing data as it travels across networks to prevent interception and tampering. It uses up-to-date, secure libraries and protocols for encryption during data transmission. It also involves implementation of Public Key Infrastructure (PKI) for end-to-end encryption of message bodies and attachments. It prevents unauthorized interception and tampering of data during transmission, ensuring the integrity and confidentiality of data as it moves between systems or users. |
| Encryption in use | -refers to the protection of data while it is being actively processed or accessed by applications. There is use of identity management systems to verify user roles and identities. Implement conditional access controls based on user roles and other parameters to restrict access to specific functionality. It helps in maintaining data security and confidentiality during processing and access, reducing the risk of exposure or unauthorized manipulation of sensitive information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | -the process of verifying a user's identity based on valid credentials, such as a username and password. To do so, authenticate users using secured local databases or external services, such as AWS, and prefer well-established protocols for authentication. Use multi-factor authentication (MFA) where possible to enhance security. Authentication ensures that only authorized individuals can access systems and resources, thereby reducing the risk of unauthorized access and enhancing overall security. |
| Authorization | -determines which resources and operations a user is permitted to access after successful authentication. It can be done by implementing role-based access control (RBAC) or attribute-based access control (ABAC) to define and enforce user permissions. Can also be done through regular reviewing and updating user roles and access levels based on changes in job functions or organizational needs. It protects resources from unauthorized access and misuse by ensuring that users have appropriate permissions. Helps maintain security by enforcing access controls based on user roles and responsibilities. |
| Accounting | -involves tracking and recording user activities and system operations to ensure compliance and detect any anomalies or unauthorized actions. It can be done by monitoring user activities, including logins, resource usage, and any changes to the database. Track addition of new users, changes in user access levels, and files accessed by users. It provides visibility into user actions and system changes, aiding in the detection of potential security incidents and ensuring accountability. It also helps in forensic investigations and compliance with regulatory requirements. |

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

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

### Map the Principles

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

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

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

| **Coding Standard** | **Principle(s)** | **Explanation** |
| --- | --- | --- |
| STD-001- CPP: Do not cast to an out-of-range enumeration value | Principle 8: Avoid Undefined Behavior  Principle 9: Enforce Correctness  Principle 10: Use Language Features Correctly | Casting to an out-of-range enumeration value can lead to undefined behavior, such as unpredictable program crashes or incorrect logic.  This standard ensures that only valid enumeration values are used, which enforces correctness in the code.  Proper usage of enumeration types as defined by the language prevents errors and misuse. This standard promotes correct and predictable behavior by following language rules regarding enumeration values. |
| STD-002- CPP: Use valid references, pointers, and iterators to reference elements of a container | Principle 4: Handle Errors and Exceptions  Principle 5: Minimize Attack Surface  Principle 8: Avoid Undefined Behavior  Principle 10: Use Language Features Correctly | Using valid references, pointers, and iterators helps to prevent runtime errors and exceptions that arise from accessing invalid or out-of-bounds container elements.  Ensuring that only valid references, pointers, and iterators are used reduces the potential for security vulnerabilities associated with accessing incorrect memory locations or corrupting data.  This standard helps to prevent undefined behavior by ensuring that references, pointers, and iterators are always valid when accessing container elements.  Proper use of references, pointers, and iterators in line with container requirements ensures that language features are employed correctly and effectively, avoiding pitfalls associated with incorrect usage. |
| STD-003- CPP: Do not attempt to create a std::string from a null pointer | Principle 2: Fail Safely  Principle 4: Handle Errors and Exceptions  Principle 5: Minimize Attack Surface  Principle 8: Avoid Undefined Behavior | Ensuring that a std::string is not created from a null pointer prevents undefined behavior or runtime errors.  By avoiding the creation of std::string from a null pointer, the code reduces the likelihood of encountering errors or exceptions related to invalid or corrupted string data.  Preventing the creation of std::string from null pointers reduces potential vulnerabilities related to memory access and invalid data handling. This minimizes the risk of security exploits that could arise from mishandled string operations.  This standard ensures that string operations are conducted with valid pointers, thus avoiding scenarios that could lead to undefined behavior. This adherence to defined operations promotes overall code reliability and correctness. |
| STD-004- CPP: Do not store already-owned pointer value in an unrelated smart pointer | Principle 1: Ensure Confidentiality  Principle 2: Fail Safely  Principle 4: Handle Errors and Exceptions  Principle 5: Minimize Attack Surface | Proper management of pointers ensures that sensitive data is not inadvertently exposed or corrupted.  Correctly managing pointer ownership prevents memory corruption and ensures the system handles pointers in a predictable manner.  Avoiding improper pointer management helps in preventing exceptions or errors related to memory access. It ensures that resource management issues do not lead to unhandled exceptions or application failures.  By avoiding incorrect pointer management, the code reduces potential vulnerabilities that could be exploited by attackers, such as those leading to memory corruption or unauthorized access. |
| STD-005- CPP: Properly deallocate dynamically allocated resources | Principle 1: Ensure Confidentiality  Principle 2: Fail Safely  Principle 4: Handle Errors and Exceptions  Principle 5: Minimize Attack Surface | Properly managing dynamic memory helps prevent potential data leakage or exposure. If resources are not deallocated properly, sensitive information might persist longer than intended, increasing the risk of unauthorized access.  Ensures that dynamic memory is correctly managed to prevent memory leaks and potential crashes, which could lead to system instability or unpredictable behavior. This contributes to system robustness and safe operation.  Proper deallocation of resources is essential in managing errors and exceptions gracefully. It prevents resource leaks which could cause the application to fail in an uncontrolled manner, affecting reliability.  By ensuring that dynamically allocated memory is properly freed, the code reduces potential vulnerabilities associated with memory management, such as buffer overflows or heap corruption, which can be exploited by attackers. |
| STD-006- CPP: Use a static assertion to test the value of a constant expression | Principle 2: Fail Safely  Principle 5: Minimize Attack Surface  Principle 9: Use Least Privilege | Using static assertions ensures that certain conditions or values are checked at compile-time, catching issues early and preventing runtime failures that could lead to unsafe conditions or security vulnerabilities.  By validating constant expressions at compile-time, static assertions help to ensure that only valid and expected values are used, reducing the potential for unexpected behavior or exploitation due to incorrect values.  Ensures that constant values are validated and constrained, reducing the risk of errors or misuse that could arise from inappropriate values. |
| STD-007- CPP: Handle all exceptions thrown before main() begins executing | Principle 1: Complete Mediation  Principle 3: Fail Securely  Principle 8: Least Common Mechanism | Ensures that all exceptions are handled effectively before the application starts, preventing any unhandled exceptions from compromising security or application stability.  Addresses potential issues by handling exceptions early in the program's lifecycle, minimizing the risk of security breaches or unexpected failures due to unhandled exceptions.  Reduces the risk of shared resource conflicts by ensuring that exceptions are handled properly and consistently across the program's initialization phase. |
| STD-008- CPP: Do not alternately input and output from a file stream without an intervening positioning call. | Principle 2: Economy of Mechanism  Principle 3: Fail Securely  Principle 6: Defense in Depth | Ensures simplicity by avoiding complex and error-prone file operations, which improves code clarity and maintainability.  Maintains data integrity by preventing data corruption or unexpected behavior through proper file positioning.  Adds an extra layer of protection by ensuring correct file stream handling, which enhances overall system robustness and security. |
| STD-009- CPP: Do not invoke virtual functions from constructors or destructors. | Principle 2: Separation of Duties  Principle 5: Secure by Design  Principle 6: Incident Handling | Involves dividing responsibilities to avoid conflicts of interest and ensure that tasks are performed correctly. By not invoking virtual functions from constructors or destructors, you prevent unintended interactions or side effects that might occur due to incomplete or inconsistent object states.  Designing secure systems involves anticipating potential problems and ensuring that design choices do not introduce vulnerabilities. Invoking virtual functions during construction or destruction can lead to undefined behavior or security issues, as the object might not be fully initialized or might be in an inconsistent state.  Effective incident handling involves anticipating and mitigating potential issues that could lead to system failures or vulnerabilities. By avoiding the invocation of virtual functions in constructors or destructors, you minimize the risk of runtime errors or unexpected behavior that could result in incidents or failures. |
| STD-010- CPP: Value returning functions must return a value from all exit paths. | Principle 1: Least Privilege  Principle 3: Fail Securely  Principle 5: Secure by Design | This principle involves ensuring that components of a system have only the minimum privileges necessary to perform their tasks. By ensuring that value-returning functions return a value from all exit paths, you avoid situations where functions might return undefined or default values.  Failing securely means that the system should handle failures in a way that minimizes security risks and maintains integrity. If a value-returning function does not cover all exit paths, it might lead to scenarios where the function does not return a valid value, potentially causing unexpected behavior or security vulnerabilities.  Secure by design involves building security into the system from the start. This principle ensures that all possible execution paths are accounted for, preventing issues that could arise from unhandled cases or incomplete logic. |

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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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