**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input data validation is a crucial process in software development that involves checking and verifying data that is entered into a system to ensure they meet specified criterion and are free from errors. This validation ensures data accuracy, integrity, and security before being processed. Data validation helps developers prevent issues like data corruption, security vulnerabilities, and other system malfunctions which may arise from invalid or malicious inputs. Some validation techniques are range checking, format checking, and length checking. These techniques help enforce data integrity and enhance the reliability of the system. Proper input validation is imperative in building robust, efficient, and secure applications. |
| 1. Heed Compiler Warnings | Heeding compiler warnings is a crucial software development practice which involves paying attention to and addressing warnings generated by the compiler during the code compilation process. Compiler warnings highlight potential issues, such as syntax errors, unused variables, or type mismatches. These issues might not prevent the program from compiling but could lead to runtime errors or other unexpected behavior. Ignoring the compiler warnings can result in faulty and inefficient code. Potential bugs can be caught early in the development process when addressing the compiler warnings, improve the quality of the code, and enhance overall system reliability. Heeding compiler warnings is a proactive approach in maintaining clean and robust code, as a result, building reliable software. |
| 1. Architect and Design for Security Policies | Architecting and designing for security policies is a critical part of developing robust and secure software applications. This process involves considering and incorporating security measures into the architectural and design phases of a software application project. This process defines and implements policies to govern access controls, data encryption, authentication processes and mechanisms, and other security approaches. Developers can implement architecture and design for security policies to create a strong foundation to secure sensitive information and prevent unauthorized access or other malicious activities. This is a proactive approach which ensures that security is an integral part of the software structure. A detailed security architecture and design works as the foundation for effective implementation and enforcement of security policies throughout the development lifecycle. |
| 1. Keep It Simple | Keeping it simple is a fundamental software development principle which advocates for simplicity and clarity in application design and implementation. This principle encourages developers to avoid unnecessary code complexities, intricate structures, or confusing and convoluted solutions. This principle advocates straight forward, easy to understand, and easily maintainable code, which can be easy to develop, understand and maintain. This principle also allows for better code readability and reduces the likelihood of introducing errors and bugs, and also facilitating collaboration among team members. This principle also helps developers to create efficient and reliable software applications. In software development, keeping it simple is a virtue that leads to quality and efficient code, and improved results. |
| 1. Default Deny | Default deny is a security principle that establishes a baseline approach in which access is denied by default. This principle only allows explicit instances which are permitted. This principle, by default blocks permission to users, programs, and processes to other processes and resources, unless specifically permitted. This minimizes the attack surface and reduces the risk of unauthorized access. This principle also lets only whitelisted code to be executed. This helps prevent malware and other malicious code execution. Default deny principle, although complex in implementation, provides increased security. This principle is less flexible, but it helps reduce the risk of errors. This principle should be used in systems handling sensitive data, and systems that are heavily regulated and systems with critical infrastructure. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege is a security concept that advocates granting users or systems only the minimum level of access or permissions necessary to perform their described roles. This principle helps minimize the potential impact of security breaches by restricting users, processes, and systems to the bare minimum privileges that are required for them to perform their functions. The principle of least privilege helps reduce the attack surface. Through the implementation of this principle developers can mitigate the risks associated with unauthorized access, limit the possibilities for unintended actions or system compromises, and enhance the overall system security. This principle is imperative for maintaining confidentiality, integrity, and availability of the system and its resources. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems is a critical practice that involves validating and cleaning data before transferring the data to external systems. This process helps prevent security vulnerabilities, data corruption, or any other unintended consequences that may come up from the introduction of malicious or improper data. Sanitization includes validating input to adhere to expected formats, removing any potentially harmful characters, and ensuring data integrity. Developers, through the implementation of this principle, can secure their applications against security threats like injection attacks or data manipulation. This is a particularly important principle especially when an application is interacting with an external API, databases, or any other third-party systems. This principle helps developers build more robust and secure applications. |
| 1. Practice Defense in Depth | Defense in depth is a comprehensive and layered approach to security in software development. This principle emphasizes multiple layers of defense and security mechanisms to protect against various potential threats and attacks. This policy involves the implementation of a combination of security measures, including encryption, access controls etc. This principle focuses on a multi-layered defense structure, where if one layer is breached, other security layers are in place to mitigate the impact and prevent further compromise. This principle helps developers build resilient systems that can withstand a diverse range of security challenges, reducing the risk of threats and attacks. Practicing Defense in Depth is a proactive approach to secured development ensuring a robust and adaptable security posture. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques ensure the delivery of reliable, high-quality products. This is a crucial principle in software development where systems have to go through systematic processes and methodologies that focus on preventing, identifying, and addressing defects or issues through the development lifecycle. Techniques like, automated testing, code reviews, static analysis and continuous integration are key roles in maintaining software quality. These techniques allow rapid code evaluation, facilitate collaboration and catch potential issues early on in the development process, as well as help identify code vulnerabilities without executing the program. Applying quality assurance techniques ensure that software applications meet specified standards, functions as intended, and provide positive user experiences. |
| 1. Adopt a Secure Coding Standard | A secure coding standard is a set of guidelines and best practices established to promote the development of secure software applications. These standards define a set of rules and recommendations that developers should follow during the coding process to minimize the risk of vulnerabilities and security threats. Secure coding standards address several aspects of software development, including input validation, authentication, data handling, and error handling. Developers, by adhering to these standards, can reduce the possibilities of security issues like buffer overflows, injection attacks, and improper authentication. The standards help create a solid foundation for software applications that prioritize security from the early stages of development, protecting sensitive data, and enhancing the overall security of the application against potential threats. |

### 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** |
| --- |
| Check if the value is within the enum value range |
| num EnumType  {  One,  Two,  Three,  };  void func(int numBer)  {  EnumType eVar = static\_cast<EnumType>(numBer);    If(eVar < One || eVar > Three)  {  }  } |

| **Compliant Code** |
| --- |
| Check value of enum before conversion to ensure the output is not an error. |
| enum EnumType  {  One,  Two,  Three,  };  void func(int numBer)  {  If(eVar < One || eVar > Three)  {  }  EnumType eVar = static\_cast<EnumType>(numBer);  } |

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

| **Principles(s):** Keep it Simple: To prevent complications or errors in calculations, it is crucial to implement and allocate integers/buffers in a straightforward, concise and simple manner. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS – Studio | 7.28 | V1016 |  |
| PRQA C/C++/Helix QAC | 3.2 | 3013 |  |
| Axivion Bauhaus Suite | 6.9.0 | CertC++ - INT50 |  |
| Parasoft C/C++ | 2023.2 | 3013 | An expression with enum underlying type will only have values corresponding to the enumerators of the enumeration. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Use the correct reference types or pointers. |

| **Noncompliant Code** |
| --- |
| Function designed to modify the value of an integer. When trying to pass a constant variable (consValue) reference, it results in a compiler error as the reference type in the function parameter is not compatible with a constant variable. |
| #include <iostream>  void updateValue(int& numRef)  {  // Incorrect usage of reference type  numRef = 10; // This modifies the original value outside the function  }  int main()  {  const int constValue = 5;  // Attempting to pass a constant variable by reference  updateValue(constValue); // Compiler error: invalid initialization of reference of type 'int&' from expression of type 'const int'  return 0;  } |

| **Compliant Code** |
| --- |
| The updateValue function below accepts a non-constant reference. This allows for the modification of the value of the variable passed to it. The main function passes a non-constant variable (value) to the updateValue function and printing the updated value. |
| #include <iostream>  void updateValue(int& numRef)  {  // Corrected to accept a constant reference  numRef = 10; // This modifies the original value outside the function  }  int main()  {  int value = 5;  // Passing a non-constant variable by reference  updateValue(value);  std::cout << "Updated value: " << value << std::endl; // Output: Updated value: 10  return 0;  } |

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

| **Principles(s):** Architect and Design for Security Policies: Carefully designing the structure and outlining how programs utilize memory, common errors can be proactively prevented and enhance security. Ensuring that elements are accessed in the correct manner helps avoid vulnerabilities and maintains adherence to security protocols. Therefore, this fortifies the program against potential risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.10 | Overflow\_unpon\_dereference |  |
| Helix QAC | 2021.3 |  |  |
| Parasoft C/C++ test | 2023.2 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| PVS-Studio | 7.24 | V783 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not make a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| The example below a std::string is constructed using a null pointer (nullPointer). This results in undefined behavior as std::string constructor expects a null-terminated string, and dereferencing a null pointer to create the string is not allowed. Any subsequent attempt to access the length of the constructed std::string also end in undefined behavior. |
| #include <iostream>  #include <string>  int main()  {  char\* nullPointer = nullptr;  // Attempting to construct a std::string from a null pointer  std::string strFromNull(nullPointer); // This leads to undefined behavior  // Attempting to use the std::string after construction  std::cout << "Length of string: " << strFromNull.length() << std::endl; // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| The corrected compliant code shows a valid non-null pointer(validString) is used to construct the std::string. This ensures compliance with the proper usage of the std::string constructor. This avoids any undefined behavior associated with constructing a std::string from a null pointer. |
| #include <iostream>  #include <string>  int main()  {  const char\* validString = "Hello, World!";  // Constructing std::string with a valid non-null pointer  std::string strFromNonNull(validString);  // Using the std::string after construction  std::cout << "Length of string: " << strFromNonNull.length() << std::endl;  return 0;  } |

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

| **Principles(s):** Input Validation: Input validation is crucial in preventing buffer overflow, null pointers, and unpredictable outcome. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.3 |  |  |
| Astree | 23.10 | Assert\_failure |  |
| Parasoft C/C++ test | 2023.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | The smart pointer value should not be stored in a pre-owned pointer. |

| **Noncompliant Code** |
| --- |
| A std::unique\_ptr is used to manage the ownership of a raw pointer (rawPointer). The ownership is then transferred to ownedPointer and attempting to access the value through the raw pointer afterward leads to undefined behavior. |
| #include <iostream>  #include <memory>  int main()  {  int\* rawPointer = new int(42);  // Attempting to store the value in a smart pointer, leading to ownership issues  std::unique\_ptr<int> ownedPointer(rawPointer); // Non-compliant: ownership transferred to ownedPointer  // Accessing the raw pointer after ownership transfer  std::cout << "Value: " << \*rawPointer << std::endl; // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| The following block of code is compliant, where the ownership of dynamic memory is directly transferred to the std::unique\_ptr during its construction. The value is accessed through the smart pointer. This avoids any issues related to accessing the raw pointer after ownership transfer. This compliant code block ensures proper memory management and best practices. |
| #include <iostream>  #include <memory>  int main()  {  // Creating a unique\_ptr and transferring ownership  std::unique\_ptr<int> ownedPointer(new int(42));  // Accessing the value through the smart pointer  std::cout << "Value: " << \*ownedPointer << std::endl;  return 0;  } |

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

| **Principles(s):** Input Validation: Validating input data serves as a proactive measure against malware infiltration into the system and ensure the safety of the data being processed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.3 |  |  |
| Astree | 23.10 | Dangling\_pointer\_use |  |
| PVS – Studio | 7.24 | V1006 |  |
| Parasoft C/C++ test | 2023.2 | CERT\_CPP-MEM56-a | Do not store a pre-owned pointer value in an unrelated smart pointer |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Deallocate all allocated resources properly. |

| **Noncompliant Code** |
| --- |
| An integer array is dynamically allocated using new int[5]. The crucial step of deallocating the allocated memory using delete[] is missing. This leads to memory leak. Memory leaks happen when dynamically allocated resources are not properly freed. This causes the program to consume more memory over time and potentially leading to underperformance. |
| #include <iostream>  int main()  {  // Allocating memory for an integer array  int\* dynamicArray = new int[5];  // Performing some operations with the allocated memory  for (int i = 0; i < 5; ++i)  {  dynamicArray[i] = i \* 2;  }  // Oops! Forgot to deallocate the allocated memory  // delete[] dynamicArray; // This line is missing, leading to a memory leak  // Accessing the memory (potentially causing issues)  for (int i = 0; i < 5; ++i)  {  std::cout << dynamicArray[i] << " ";  }  return 0;  } |

| **Compliant Code** |
| --- |
| To make the above code compliant, delete[] dynamicArray; line has been added to properly deallocate the memory allocated for the integer array. This ensures that the code is compliant and preventing memory leaks. Attempting to access the memory before deallocation will result in undefined behavior. The last loop is included to access the memory after deallocation. |
| #include <iostream>  int main()  {  // Allocating memory for an integer array  int\* dynamicArray = new int[5];  // Performing some operations with the allocated memory  for (int i = 0; i < 5; ++i)  {  dynamicArray[i] = i \* 2;  }  // Deallocating the allocated memory to prevent memory leak  delete[] dynamicArray;  // Accessing the memory after deallocation  for (int i = 0; i < 5; ++i)  {  std::cout << dynamicArray[i] << " "; // This line will cause issues; it's just to demonstrate that the memory is deallocated  }  return 0;  } |

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

| **Principles(s):** Keep it Simple: Memory management aligns with keeping things simple by preventing unnecessary use of memory bits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2023.4 | CL.FFM.ASSIGNFM  CL.FFM.COPY  CL.FMM  FMM.MIGHT  FMM.MUST  FNH.MIGHT  FNH.MUST  FUM.GEN.MIGHT  FUM.GEN.MUST  UNINIT.CTOR.MIGHT |  |
| CodeSonar | 8.0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM | Free non-heap variable  Double free  Type mismatch |
| Clang | 17 | Clang-analyzer-cplusplus.NewDeleteLeaks-Wmismatched-new-delete-clang-analyzer-unix.MismatchedDeallocator |  |
| Astree | 23.10 | Invalid\_dynamic\_memory\_allocation\_dangling\_pointer\_use |  |
| LDRA tool suite | 9.8 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D | Partial implementation |
| Parasoft Insure++ |  |  | Detection at runtime |
| PVS-Studio | 7.24 | V515,V554, V611, V701, V748, V773 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | When testing constant expressions, they need to be tested with static assertion. |

| **Noncompliant Code** |
| --- |
| In the following example, a constant expression size, is declared with a value of 5. The assert() statement attempts to test if size is greater than 0. However, using assert() for constant expressions is non-compliant as assert() is meant for runtime checks. Constant expressions should be validated using static assertions. |
| #include <cassert>  #include <iostream>  int main()  {  constexpr int size = 5;  // Non-compliant: Using assert to test constant expressions  assert(size > 0); // This will result in a compilation error  std::cout << "Used assert." << std::endl; // This line won't be reached  return 0;  } |

| **Compliant Code** |
| --- |
| The following code block is a correction of the above code block, where static\_assert is used to test the constant expression at compile time, providing an error message if assertion fails. |
| #include <iostream>  int main()  {  constexpr int size = 5;  // Compliant: Using static\_assert to test constant expressions  static\_assert(size > 0, "Size must be greater than 0.");  std::cout << "Used static\_assert." << std::endl;  return 0;  } |

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

| **Principles(s):** Employ effective QA techniques: Detecting issues early prevents problems from escalating. Utilizing static assertions offer a means of QA eliminating defects.  Keeping it Simple: Avoid unnecessary program complexity. Implement assertions simply to uncover hidden problems within the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.8 | 44S | Full Implementation |
| Éclair Bug finder | 1.2 | CC2.DCL03 | Full Implementation |
| Axivion Bauhaus Suite | 7.2.0 | CERTC-DCL03 |  |
| CodeSonar | 8.0 |  | A custom check can be implemented that reports uses of the assert() macro |
| Clang | 17 | Misc-static-assert |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions before executing. |

| **Noncompliant Code** |
| --- |
| The performOperation function throws a std::runtime\_error exception in the main function. If an exception is thrown during the performOperation call, the program will terminate abruptly, and any code after the exception will not be reached or executed. |
| #include <iostream>  void performOperation()  {  throw std::runtime\_error("An exception occurred!"); // Non-compliant: Exception is not caught  }  int main()  {  performOperation(); // Exception is thrown but not caught  // Code here won't be executed if the exception is thrown  std::cout << "Code after the exception." << std::endl; // This line won't be reached  return 0;  } |

| **Compliant Code** |
| --- |
| To make the above code compliant, the exception should be caught using try-catch block. The try-catch block is used to catch the exception thrown by performOperation. This allows the program to handle the exception properly and continue execution. |
| #include <iostream>  void performOperation()  {  throw std::runtime\_error("An exception occurred!");  }  int main()  {  try  {  performOperation(); // Exception is thrown  }  catch (const std::exception& ex)  {  std::cerr << "Caught exception: " << ex.what() << std::endl;  }  // Code here will be executed even if an exception is thrown and caught  std::cout << "Code after the exception handling." << std::endl;  return 0;  } |

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

| **Principles(s):** Architect and Design for Security Policies – Exceptions can be caught with a well-designed application.  Keep it simple: Crafting clear and concise code facilitates ease of understanding, therefore, contributing to the effective validation of assertions when implemented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.10 | Potentially-throwing-static-initialization | Checked partially |
| Parasoft C/C++ test | 2023.2 |  | Exceptions can be raised only after start-up and before termination of the program. |
| Helix QAC | 2021.3 |  |  |
| Axivion Bauhaus Suite | 7.2.0 | CERTC++ ERR58 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Do not invoke virtual functions from constructors or destructors. |

| **Noncompliant Code** |
| --- |
| In the example below, both the base class Base and derived class Derived have virtual functions (perform operations and performCleanup). These are called from their respective constructors and destructors. Calling virtual functions from constructors and destructors can lead to unexpected behavior, as the dynamic object type may not be fully constructed or may already have been destroyed. This can lead to calling the wrong implementation of the virtual function or accessing resources that are not valid anymore. |
| #include <iostream>  class Base  {  public:  Base()  {  // Non-compliant: Virtual function called from constructor  performOperation();  }  virtual ~Base()  {  // Non-compliant: Virtual function called from destructor  performCleanup();  }  virtual void performOperation()  {  std::cout << "Base class operation." << std::endl;  }  virtual void performCleanup()  {  std::cout << "Base class cleanup." << std::endl;  }  };  class Derived : public Base  {  public:  Derived()  {  // Non-compliant: Virtual function called from constructor  performOperation();  }  ~Derived() override  {  // Non-compliant: Virtual function called from destructor  performCleanup();  }  void performOperation() override  {  std::cout << "Derived class operation." << std::endl;  }  void performCleanup() override  {  std::cout << "Derived class cleanup." << std::endl;  }  };  int main()  {  Derived derivedObject; // This will lead to non-compliant behavior  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version of the above code, the virtual functions performOperations, and performCleanup are not called from the constructors or destructors of the classes. These operations are performed in the main function after the object is properly constructed. This helps to avoid potential issues that are associated with calling virtual functions during object constructions or destruction. |
| #include <iostream>  class Base  {  public:  Base()  {  // Compliant: Avoid calling virtual functions from constructor  }  virtual ~Base()  {  // Compliant: Avoid calling virtual functions from destructor  }  virtual void performOperation()  {  std::cout << "Base class operation." << std::endl;  }  virtual void performCleanup()  {  std::cout << "Base class cleanup." << std::endl;  }  };  class Derived : public Base  {  public:  Derived()  {  // Compliant: Avoid calling virtual functions from constructor  }  ~Derived() override  {  // Compliant: Avoid calling virtual functions from destructor  }  void performOperation() override  {  std::cout << "Derived class operation." << std::endl;  }  void performCleanup() override  {  std::cout << "Derived class cleanup." << std::endl;  }  };  int main()  {  Derived derivedObject; // This is now compliant  // Perform operations after the object is fully constructed  derivedObject.performOperation();  derivedObject.performCleanup();  return 0;  } |

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

| **Principles(s):** Keep the Simple: Design and code simplicity and design with composition will help prevent vulnerabilities related to invoking virtual functions from constructors and destructors. Instead of relying heavily on inheritance, using composition to achieve the desired behavior. Having separate objects for specific functions, and initializing or destroying them with the constructor or destructor. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.10 | Virtual-call-in-constructor  Invalid\_function\_pointer | Checked fully |
| Helix QAC | 2021.3 |  |  |
| LDRA tool suite | 9.8 | 467S, 92D | Full implementation |
| PVS-Studio | 7.24 | Virtual-call-in | Checked fully |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output (IO) | [STD-009-CPP] | Input and output from a file stream with an intervening positioning call. |

| **Noncompliant Code** |
| --- |
| In the example below, std::fstream is opened for both input and output operations on a file named “example.txt”. the code alternately attempts to read from the file without checking its current position and then writes to the file without intervening positioning call. This can lead to unexpected behavior as there is not guarantee that the file’s current position between input and output operations. |
| #include <iostream>  #include <fstream>  int main()  {  std::fstream file("example.txt", std::ios::in | std::ios::out | std::ios::trunc);  if (!file.is\_open())  {  std::cerr << "Error opening the file." << std::endl;  return 1;  }  // Non-compliant: Alternately reading and writing without positioning  int value;  file >> value; // Attempting to read from an empty file  file << "Hello, World!"; // Writing to the file without positioning  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version of the above code, seekg() is used to set the file position for reading at the beginning before attempting to read from the file. Likewise, seekp() is used to set the file position for writing at the end before writing to the file. This ensures that the file position is explicitly managed, avoiding unexpected behavior when alternating between reading and writing operations. |
| #include <iostream>  #include <fstream>  int main()  {  std::fstream file("example.txt", std::ios::in | std::ios::out | std::ios::trunc);  if (!file.is\_open())  {  std::cerr << "Error opening the file." << std::endl;  return 1;  }  // Compliant: Using positioning calls to set the file position  int value;  file.seekg(0); // Set the file position for reading at the beginning  file >> value; // Read from the file  file.seekp(0, std::ios::end); // Set the file position for writing at the end  file << "Hello, World!"; // Write to the file  return 0;  } |

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

| **Principles(s):** Input Validation: validate input data before using it in file operations. Ensure that user input or external data is within expected ranges and formats to prevent potential vulnerabilities like buffer overflows.  Positioning Calls: It is important to be cautious when using positioning calls. It is important to understand the file format and structure, and it is important to be mindful of potential impact on read and write operations.  Error Handling: implementing robust error handling mechanisms to handle potential failures during file operations. Exceptions or return codes should be used to convey error conditions and provide meaningful error messages. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.10 |  | No explicit checker |
| CodeSonar | 8.0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2023.2 | CERT\_C-FIO42-a | Free resources |
| Polyspace Bug Finder | 2023A | CERT C: Rule FIO42-C | Check resource leak |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-010-CPP] | Use of valid iterator ranges |

| **Noncompliant Code** |
| --- |
| In the example below, begin iterator is set to point to the third element of the vector, and the end iterator is set to an invalid position by subtracting 1 from the beginning iterator. The subsequent attempt to iterate over this invalid range can lead to undefined behavior, as the end iterator is positioned before the beginning iterator. |
| #include <iostream>  #include <vector>  int main()  {  std::vector<int> numbers = {1, 2, 3, 4, 5};  // Non-compliant: Invalid iterator range  auto begin = numbers.begin() + 2; // Iterator pointing to the third element  auto end = numbers.begin() - 1; // Invalid iterator position  // Attempting to iterate over the invalid range  for (auto it = begin; it != end; ++it)  {  std::cout << \*it << " ";  }  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version of the above code block, the end iterator is set to numbers.end(), which points to one past the last element in the vector. This ensures that the iterator range is valid and the loop iterates over the elements between begin and end. |
| #include <iostream>  #include <vector>  int main()  {  std::vector<int> numbers = {1, 2, 3, 4, 5};  // Compliant: Valid iterator range  auto begin = numbers.begin() + 2; // Iterator pointing to the third element  auto end = numbers.end(); // Iterator pointing to one past the last element  // Iterating over the valid range  for (auto it = begin; it != end; ++it)  {  std::cout << \*it << " ";  }  return 0;  } |

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

| **Principles(s):** Use Container Member Functions: When required, it is important to use container member functions like begin() and end(), instead of explicit iterator declarations. This ensures that we get valid iterators without having to account for the container’s underlying details. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | Cert C++-MSC52 |  |
| Astree | 23.10 | Return implicit | Fully checked |
| CodeSonar | 8.0 | LANG.STRUCT | Missing return statement |
| PVS-Studio | 7.24 | V591 |  |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Probable | High | P6 | 2 |
| STD-003-CPP | High | Likely | Medium | P18 | 1 |
| STD-004-CPP | High | Likely | Medium | P18 | 1 |
| STD-005-CPP | High | Likely | Medium | P18 | 1 |
| STD-006-CPP | Low | Unlikely | High | P1 | 3 |
| STD-007-CPP | Low | Likely | Low | P9 | 2 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | 3 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | 3 |
| STD-010-CPP | Medium | Unlikely | High | P8 | 2 |

### 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 | At-rest encryption is designed to prevent unauthorized access to unencrypted data by ensuring that data is encrypted while stored on disk. If an adversary acquires a hard drive containing encrypted data but lacks the encryption keys, they must overcome the encryption to decipher and access the data. |
| Encryption at flight | This is the process of encrypting data during its transmission. In certain applications, like remote replication, data may remain unencrypted while stored on drive arrays (at rest), yet it is encrypted during transmission to ensure security. |
| Encryption in use | Compromising data in use allows for unauthorized access to both encrypted data at rest and data in motion. For example, an individual with access to random access memory can scrutinize the memory to identify the encryption key associated with data at rest. Once in possession of this encryption key, they can subsequently decrypt the encrypted data at rest. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the verification process confirming a user’s identity and granting access to the system. This involves validating user credentials, such as login and password information, to enable access to specific system components. Modern approaches include two-step authentication or multi-tier authentication for enhanced security. |
| Authorization | Authorization pertains to the extent of a user’s access within the system, encompassing permissions to read, create, delete, or modify files in the database. It also dictates whether a user has the authority to add or remove files and users within the system. |
| Accounting | Accounting involves the continuous monitoring of a user’s activities within the system, tracking their level of access. This process records details such as which systems were accessed, which actions performed during access, and the user responsible for initiating the system access. |

**\***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 the 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 |  |
| 1.1 | 01/27/2024 | First Draft | Abu S. Alam |  |
| 1.2 | 02/18/2024 | Final Draft | Abu S. Alam |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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