**Green Pace Developer: Security Policy Guide**



# 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 is instrumental in ensuring the security and integrity of data inputs, thereby shielding the system from potential harm. By verifying that the data being input is secure and devoid of corruption, organizations can mitigate the risk of malicious attacks or inadvertent errors that could compromise system functionality or compromise sensitive information. Upholding this principle is essential for maintaining the reliability and trustworthiness of the system, as it establishes a robust barrier against threats and vulnerabilities stemming from compromised or tainted data inputs. |
| 1. Heed Compiler Warnings | Failure to address compiler warnings can leave software vulnerable to exploitation by malicious actors. It is imperative to promptly resolve these warnings either during the debugging process or by documenting them for future attention. By neglecting to address compiler warnings, developers risk overlooking potential vulnerabilities that attackers could leverage to compromise the integrity and security of the software. Therefore, proactive mitigation of compiler warnings is crucial for fortifying the codebase and minimizing the risk of security breaches. |
| 1. Architect and Design for Security Policies | Building a secure system that aligns with established security policies and incorporates suitable variables is paramount for safeguarding sensitive data and protecting against cyber threats. Moreover, it's vital to ensure that the system's design and implementation meet the specific security needs outlined by the organization's business requirements. By integrating security considerations from the outset and aligning them with business objectives, organizations can establish a robust foundation for mitigating risks and fostering a secure operational environment. This proactive approach not only enhances data protection but also bolsters overall resilience against potential security breaches and vulnerabilities. |
| 1. Keep It Simple | It's imperative to always prioritize simplicity and conciseness in your codebase. By adhering to this principle, developers can minimize the chances of introducing errors and improve overall code readability and maintainability. Overcomplicating code not only makes it more challenging to understand but also increases the likelihood of bugs and vulnerabilities. Therefore, by striving for simplicity, developers can enhance the robustness and efficiency of their code while reducing the potential for errors, ultimately leading to more reliable and maintainable software solutions. |
| 1. Default Deny | Restricting access to files exclusively to individuals with the requisite system permissions is fundamental for safeguarding authorized data. By implementing strict access controls, organizations can prevent unauthorized users from compromising sensitive information. Equally important is the timely provision of permissions to users when access to specific data is necessary for their tasks. This balanced approach ensures that authorized personnel can access the required resources without compromising security protocols. By adhering to this principle, organizations can effectively mitigate the risk of data breaches and uphold the confidentiality and integrity of their information assets. |
| 1. Adhere to the Principle of Least Privilege | Limiting users' access to only necessary information is crucial for maintaining system security and integrity. By implementing controls that restrict user privileges, organizations can mitigate the risk of unauthorized data manipulation or exposure. This approach not only minimizes the likelihood of accidental or intentional misuse of sensitive information but also enhances overall system reliability. By adhering to the principle of least privilege, organizations can effectively mitigate security risks and ensure that users only interact with data and functionalities essential to their roles and responsibilities. Thus, fostering a culture of limited control helps safeguard against potential threats and reinforces the security posture of the system. |
| 1. Sanitize Data Sent to Other Systems | Eliminating redundant data from the system is essential to minimize security vulnerabilities, as hackers can exploit such data to glean valuable insights or sensitive information. By systematically purging redundant data, organizations can streamline their data environment and reduce the potential attack surface available to malicious actors. This proactive approach ensures that only necessary and actively utilized data remains within the system, mitigating the risk of unauthorized access or data breaches. Additionally, regular data purging practices contribute to compliance with data protection regulations and promote efficient data management practices, thereby enhancing overall system security and integrity. |
| 1. Practice Defense in Depth | Defense in Depth (DiD) is a robust security strategy that entails integrating multiple layers of protection within a system. These layers operate collaboratively to establish a formidable defense, wherein if one layer is breached or compromised, others stand ready to mitigate the threat. This approach significantly enhances the overall security posture of the system by adding redundancy and resilience to its defenses. While there is no universally defined limit to the number of layers required for optimal protection, the principle of Defense in Depth underscores the importance of diversifying security measures to address various attack vectors comprehensively. By embracing this multifaceted approach, organizations can effectively thwart potential threats and safeguard sensitive data and resources against malicious actors. |
| 1. Use Effective Quality Assurance Techniques | Efficient, regular, and effective testing is paramount to maintaining the integrity and functionality of software systems. By embracing this approach and conducting testing on a consistent basis, developers can minimize the likelihood of errors and vulnerabilities in their code. Leveraging various testing methodologies, including engaging white-hat hackers, or performing penetration testing, further fortifies the security of the system. These proactive measures not only identify potential weaknesses but also enable timely remediation, bolstering the system's resilience against cyber threats. Therefore, prioritizing comprehensive testing practices is essential for safeguarding the reliability and security of software applications in today's dynamic digital landscape. |
| 1. Adopt a Secure Coding Standard | Establishing a secure coding standard serves as a cornerstone for instilling a culture of security consciousness within the workplace. This standard provides clear guidelines and best practices to ensure that software development processes prioritize security considerations at every step. However, the implementation of secure coding practices can differ based on the specific programming language and platform employed. Factors such as language syntax and platform architecture necessitate tailored approaches to address security vulnerabilities effectively. Despite these variations, the overarching goal remains consistent: to mitigate risks and enhance the resilience of software systems against potential threats. Therefore, organizations must adapt their strategies to accommodate the nuances of different languages and platforms while upholding the principles of secure coding. |

### 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 | Avoid casting to an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| The code example I'm presenting exhibits noncompliance as it endeavors to ascertain whether a specified value falls within the acceptable range of enumeration values. However, it does so by casting the value to the enumeration type, potentially obstructing its ability to accurately represent the provided integer value. |
| #include <iostream>  enum class EnumType {  First,  Second,  Third  };  bool isValidEnumValue(int intValue) {  int minValue = static\_cast<int>(EnumType::First);  int maxValue = static\_cast<int>(EnumType::Third);  return (intValue >= minValue && intValue <= maxValue);  }  int main() {  int valueToCheck = 1;  if (isValidEnumValue(valueToCheck)) {  std::cout << "The value is within the range of acceptable enumeration values.\n";  } else {  std::cout << "The value is not within the range of acceptable enumeration values.\n";  }  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the code ensures that a value can be accurately represented by the enumeration type before proceeding with the conversion. This precaution is taken to prevent the conversion from inadvertently representing an unspecified value. |
| #include <iostream>  enum class EnumType {  First,  Second,  Third  };  bool isValidEnumValue(int value) {  // Check if the value falls within the range of enumeration values  return (value >= static\_cast<int>(EnumType::First) &&  value <= static\_cast<int>(EnumType::Third));  }  EnumType convertToEnum(int value) {  // Check if the value can be represented by the enumeration type  if (isValidEnumValue(value)) {  return static\_cast<EnumType>(value);  } else {  // Handle error or return a default value  std::cerr << "Error: Value cannot be represented by the enumeration type.\n";  return EnumType::First; // Returning a default value  }  }  int main() {  int valueToConvert = 1;  EnumType convertedValue = convertToEnum(valueToConvert);  std::cout << "Converted enum value: " << static\_cast<int>(convertedValue) << std::endl;  return 0;  } |

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

| **Principles(s):** Unspecified values can potentially trigger a buffer overflow, allowing attackers to execute arbitrary code. However, since enumerators are seldom utilized for indexing into arrays or performing pointer arithmetic, such occurrences are more likely to cause data integrity violations rather than enabling arbitrary code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PRQA QA- C++ | 4.4 | 3013 |  |
| Helix QAC | 2023.3 |  |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ - INT50 |  |
| PVS – Studio | 7.29 | V1016 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Avoid assigning an already-owned pointer value to an unrelated smart pointer. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, two unrelated smart pointers are instantiated using the same underlying pointer value. Consequently, when the local variable is destroyed, it deallocates the pointer associated with that variable. Similarly, upon destruction of the other variable, the pointer it holds is also deallocated. This situation presents a double-free vulnerability. |
| #include <memory>  void f() {  int \*i = new int(42);  std::shared\_ptr<int> p1(i); // First smart pointer  std::shared\_ptr<int> p2(i); // Second smart pointer  // Upon exiting the function, both p1 and p2 will be destroyed  // Since they both hold the same underlying pointer, double deletion will occur  } |

| **Compliant Code** |
| --- |
| In the compliant code example provided, upon destruction of the local variable p2, the shared pointer value is decreased, yet it remains 0. Subsequently, when the local variable p1 is destroyed, the shared pointer value is decreased to 0 as well, leading to the destruction of the managed pointer. It's notable that the std::shared\_ptr objects are interrelated. In this compliant code, std::make\_shared() is utilized instead of allocating a raw pointer and then assigning its value to the variable. |
| #include <memory>  void f() {  // Using std::make\_shared to create shared pointers  auto p1 = std::make\_shared<int>(42);  auto p2 = p1; // Creating another shared pointer that shares ownership with p1    // Upon exiting the function, both p1 and p2 will be destroyed  // The managed pointer will be deallocated only after both shared pointers are destroyed  } |

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

| **Principles(s):** Utilizing invalid references, pointers, or iterators to access elements of a container leads to undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Overflow\_unpon\_dereference |  |
| Helix QAC | 2023.3 |  |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| PVS – Studio | 7.29 | V783 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Avoid creating a std::string from a null pointer. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, an std::string object is instantiated using the result of std::getenv(). However, since std::getenv() returns a null pointer upon failure in this specific scenario, this code may result in undefined behavior. |
| #include <iostream>  #include <cstdlib> // For std::getenv  #include <string>  void f() {  const char\* envVar = std::getenv("SOME\_ENV\_VARIABLE");  std::string envValue(envVar); // Creating std::string from possibly null pointer    // Further operations with envValue...  }  int main() {  f();  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant example, the return value of std::getenv() is verified for null before constructing the std::string object. |
| #include <iostream>  #include <cstdlib> // For std::getenv  #include <string>  void f() {  const char\* envVar = std::getenv("SOME\_ENV\_VARIABLE");  if (envVar != nullptr) {  std::string envValue(envVar); // Creating std::string only if envVar is not null  // Further operations with envValue...  } else {  std::cerr << "Environment variable not found or empty.\n";  // Handle error appropriately...  }  }  int main() {  f();  return 0;  } |

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

| **Principles(s):** Dereferencing a null pointer is undefined behavior, often resulting in abnormal program termination. In certain situations, it can lead to the execution of arbitrary code. The severity of this issue may vary depending on the platform's susceptibility to such exploitation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.3 |  |  |
| Astree | 22.10 | Assert\_failure |  |
| ParasoftC/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-C | Avoid modifying string literals. |

| **Noncompliant Code** |
| --- |
| In the given example, the char pointer "string" is assigned the address of a string literal. Modifying this string literal will result in undefined behavior. |
| #include <iostream>  int main() {  // Assigning the address of a string literal to a char pointer  char\* string = "Hello, world!";  // Attempting to modify the string literal  string[0] = 'h'; // This operation leads to undefined behavior  // Printing the modified string  std::cout << string << std::endl; // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| A string literal defines the initial characters in an array and its size, acting as an array initializer. It's considered poor practice to specify the size of a character array initialized with a string literal. In this example, the code will duplicate the string literal into the allocated space for the character array "string". |
| #include <iostream>  int main() {  // Defining a string literal  const char\* literal = "Hello, world!";  // Initializing a character array with a string literal  char string[] = "Hello, world!";  // Printing the addresses of the string literal and the character array  std::cout << "Address of string literal: " << static\_cast<const void\*>(literal) << std::endl;  std::cout << "Address of character array: " << static\_cast<void\*>(string) << std::endl;  return 0;  } |

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

| **Principles(s):** Passing a pointer value to a deallocation function that was not previously acquired from the corresponding allocation function results in undefined behavior, potentially leading to exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Dangling\_pointer\_use |  |
| Helix QAC | 2023.3 | [Insert text.] |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| PVS – Studio | 7.29 | V1006 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Ensure proper deallocation of dynamically allocated resources. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the local variable "space" is used as the argument for the placement of the new operator. The resulting pointer from this operation is then passed to the ::operator delete() function. This results in undefined behavior as ::operator delete() attempts to deallocate memory that was not allocated by ::operator new(). |
| #include <iostream>  void f() {  int space;  int\* ptr = new (&space) int; // Placement new using local variable "space"  ::operator delete(ptr); // Deleting memory not allocated by ::operator new  }  int main() {  f();  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to ::operator delete() is removed, and instead, s1's destructor is explicitly invoked. This is one of the rare instances where explicitly calling a destructor is justified. |
| #include <iostream>  class MyClass {  public:  ~MyClass() {  std::cout << "Destructor called\n";  }  };  void f() {  MyClass\* s1 = new MyClass;    // Do something with s1...  s1->~MyClass(); // Explicitly call the destructor  // Memory allocated by 'new' is not deallocated explicitly  }  int main() {  f();  return 0;  } |

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

| **Principles(s):** Keep It Simple: Simplicity is key in memory management, achieved by minimizing the unnecessary utilization of bits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: STR51-CPP | Examines string operations performed on a null pointer (partially addressed by the rule). |
| Astree | 22.10 | Assert\_failure |  |
| Parasoft\_V | 2023.1 | CERT\_CPP-STRS1-a | Prevent dereferencing null pointers. |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| This noncompliant code employs the assert() macro to validate a crucial property regarding a memory-mapped structure, which is vital for the correct behavior of the code. |
| #include <cassert>  struct MemoryMappedStructure {  int value;  };  void foo(const MemoryMappedStructure& obj) {  // Asserting that the value of the memory-mapped structure is within a valid range  assert(obj.value >= 0 && obj.value <= 100);    // Further code implementation...  }  int main() {  // Assuming "obj" is a memory-mapped structure  MemoryMappedStructure obj;  obj.value = 150;  foo(obj); // Call to function with memory-mapped structure    return 0;  } |

| **Compliant Code** |
| --- |
| In cases where assertions involve only constant expressions, a preprocessor conditional statement can be employed, as demonstrated in this compliant solution: |
| #include <iostream>  #define ASSERT\_CONSTEXPR(expr) \  do { \  if (!(expr)) { \  std::cerr << "Assertion failed: " #expr "\n"; \  std::terminate(); \  } \  } while(0)  struct MemoryMappedStructure {  int value;  };  void foo(const MemoryMappedStructure& obj) {  // Preprocessor conditional statement to validate the value of a constant expression  ASSERT\_CONSTEXPR(obj.value >= 0 && obj.value <= 100);    // Further code implementation...  }  int main() {  // Assuming "obj" is a memory-mapped structure  MemoryMappedStructure obj;  obj.value = 150;  foo(obj); // Call to function with memory-mapped structure    return 0;  } |

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

| **Principles(s):** Employ Effective Quality Assurance Methods – Early identification of issues helps to curb potential escalation. Utilizing static assertions furnishes developers with quality assurance and can eradicate defects accordingly.  Prioritize Simplicity – Refrain from unnecessarily complicating the program. Employ or create assertions in the most straightforward manner to uncover hidden issues within the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 | [Insert text.] |
| ECLAIR | 1.2 | CC2.DCL03 | Fully Implemented |
| LDRA | 9.7.1 | 44 S | Fully Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Ensure that all exceptions thrown before the execution of main() are properly handled. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the constructor for S may throw an exception that isn't caught when globalS is constructed during program startup. |
| #include <iostream>  class S {  public:  S() {  // Simulating a scenario where the constructor may throw an exception  throw std::runtime\_error("Exception in constructor of S");  }  };  S globalS; // Global object declaration  int main() {  std::cout << "Main function started.\n";  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, converting globalS into a local variable with static storage duration ensures that any exceptions thrown during object construction can be caught. This is because the constructor for S will be executed the first time the function globalS() is called, rather than at program startup. Implementing this solution does require the programmer to modify the source code so that previous uses of globalS are replaced by a function call to globalS(). |
| #include <iostream>  class S {  public:  S() {  // Simulating a scenario where the constructor may throw an exception  throw std::runtime\_error("Exception in constructor of S");  }  };  S& globalS() {  static S instance;  return instance;  }  int main() {  try {  // Creating globalS as a local variable with static storage duration  globalS();  } catch (const std::exception& e) {  std::cerr << "Exception caught: " << e.what() << std::endl;  }  std::cout << "Main function started.\n";  return 0;  } |

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

| **Principles(s):** Plan and Design for Security Policies – Through a carefully considered design, developers can effectively identify exceptions.  Prioritizing Simplicity – Developers who are adept at crafting straightforward and clear code simplify comprehension for others. This aids in verifying whether assertions function as intended when implemented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Potentially-throwing-staticinitialization | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.EXCP.THROW | Use of Throw |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| Parasoft | 2023.1 | CERT\_CPP\_ERR58-a | Exceptions shall be raised only after start-up and before termination of the program. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | STD-008-CPP | Avoid calling virtual functions from constructors or destructors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class `Base` attempts to manage an object's resources by calling virtual functions `seize()` in the constructor and `release()` in the destructor. However, when the derived class `Derived` is constructed and destroyed, the calls to `seize()` and `release()` in `Base` do not invoke the overridden versions in `Derived`. This leads to unexpected behavior as the base class's versions of these functions are called instead. |
| #include <iostream>  class Base {  public:  Base() {  seize(); // Attempting to seize resources in the constructor  }  virtual ~Base() {  release(); // Attempting to release resources in the destructor  }  virtual void seize() {  std::cout << "Base::seize() called\n";  }  virtual void release() {  std::cout << "Base::release() called\n";  }  };  class Derived : public Base {  public:  Derived() {  std::cout << "Derived constructor\n";  }  ~Derived() {  std::cout << "Derived destructor\n";  }  void seize() override {  std::cout << "Derived::seize() called\n";  }  void release() override {  std::cout << "Derived::release() called\n";  }  };  int main() {  Derived d;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors invoke a nonvirtual, private member function (suffixed with "mine") instead of calling a virtual function. As a result, each class assumes responsibility for acquiring and releasing its own resources. |
| #include <iostream>  class Base {  public:  Base() {  seizeMine(); // Call nonvirtual member function for seizing resources  }  ~Base() {  releaseMine(); // Call nonvirtual member function for releasing resources  }  private:  void seizeMine() {  std::cout << "Base::seizeMine() called\n";  }  void releaseMine() {  std::cout << "Base::releaseMine() called\n";  }  };  class Derived : public Base {  public:  Derived() {  std::cout << "Derived constructor\n";  }  ~Derived() {  std::cout << "Derived destructor\n";  }  private:  void seizeMine() {  std::cout << "Derived::seizeMine() called\n";  }  void releaseMine() {  std::cout << "Derived::releaseMine() called\n";  }  };  int main() {  Derived d;  return 0;  } |

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

| **Principles(s):** Prioritize Simplicity – Crafting code that is simple and straightforward yields numerous advantages. One such advantage is mitigating access to non-existent members.  Plan and Design for Security Policies – Defining explicit policies delineating security violations can underscore the consequences of accessing non-existent members. Such actions could result in unpredictable behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.UVAR | Uninitialized Variable |
| Polyspace Bug Finder | R2023b | CERT C++: OOP55-CPP | Verifies pointers accessing non-existent class members. |
| Parasoft | 2023.1 | CERT\_CPP-OOP55-a | Runtime Detection |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-009-CPP | Select a termination approach. |

| **Noncompliant Code** |
| --- |
| Calling the abort() function is discouraged because it skips performing necessary application cleanup before exiting. However, in this noncompliant code example, abort() is invoked after sending data to a descriptor. Consequently, the data may or may not have been successfully written to the file. |
| #include <iostream>  #include <cstdlib>  void sendDataToFile(const char\* data) {  // Assume writing data to a file descriptor  std::cout << "Data sent to file descriptor: " << data << std::endl;  }  void performCleanup() {  // Assume performing necessary application cleanup  std::cout << "Performing application cleanup...\n";  }  int main() {  const char\* data = "Some data";  sendDataToFile(data); // Sending data to a file descriptor  // Noncompliant: Calling abort() without performing necessary cleanup  std::abort();  // Application cleanup will not be performed if abort() is called  performCleanup();  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, the abort() function has been substituted with the exit() function. This ensures that any buffered data is flushed to the descriptor when it is correctly closed. |
| #include <iostream>  #include <cstdlib>  void sendDataToFile(const char\* data) {  // Assume writing data to a file descriptor  std::cout << "Data sent to file descriptor: " << data << std::endl;  }  void performCleanup() {  // Assume performing necessary application cleanup  std::cout << "Performing application cleanup...\n";  }  int main() {  const char\* data = "Some data";  sendDataToFile(data); // Sending data to a file descriptor  // Compliant: Using exit() instead of abort()  std::exit(EXIT\_FAILURE);  // Application cleanup will be performed after exit()  performCleanup();  return 0;  } |

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

| **Principles(s):** Employ Effective Quality Assurance Methods – Quality assurance serves as the guardian of ensuring proper element verification. Neglecting to validate elements during quality assurance can result in out-of-bounds reads/writes.  Embrace a Secure Coding Standard - Implementing secure coding standards is crucial for ensuring that elements remain within the designated range. Failure to check elements for proper range can leave the program susceptible to vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA | Buffer overflow  - Buffer underflow  - Contaminated buffer access  - Type overflow  - Type underflow |
| Astree | 22.10 | assert\_failure |  |
| Parasoft | 2023.1 | CERT\_CPP-STR53-a | Ensure that container indices fall within the valid range. |
| Polyspace Bug Finder | R2023b | CERT C++: STR53-CPP | Verifies for:  - Array access beyond bounds  - Array access using a tainted index  - Pointer dereference with a tainted offset |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Signals | STD-010-CPP | Avoid utilizing signals for indicating normal functionality. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, signals are employed to convey state changes within a multithreaded environment. |
| #include <iostream>  #include <csignal>  #include <thread>  #include <chrono>  bool state = false;  void signalHandler(int signum) {  if (signum == SIGUSR1) {  state = true;  } else if (signum == SIGUSR2) {  state = false;  }  }  void threadFunction() {  while (true) {  // Simulate thread activity  std::this\_thread::sleep\_for(std::chrono::seconds(1));  // Check state periodically  if (state) {  std::cout << "State changed: true\n";  } else {  std::cout << "State changed: false\n";  }  }  }  int main() {  // Register signal handlers  signal(SIGUSR1, signalHandler);  signal(SIGUSR2, signalHandler);  // Start a thread  std::thread t(threadFunction);  // Main thread  while (true) {  // Simulate main thread activity  std::this\_thread::sleep\_for(std::chrono::seconds(2));  // Toggle state using signals  std::raise(state ? SIGUSR2 : SIGUSR1);  }  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, utilizing conditional variables is preferred. Specifically, the compliant code employs a conditional variable from the POSIX library in this instance. |
| #include <iostream>  #include <pthread.h>  #include <unistd.h>  bool state = false;  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;  void\* threadFunction(void\*) {  while (true) {  pthread\_mutex\_lock(&mutex);  while (!state) {  pthread\_cond\_wait(&cond, &mutex);  }  std::cout << "State changed: true\n";  state = false;  pthread\_mutex\_unlock(&mutex);  }  return nullptr;  }  int main() {  pthread\_t tid;  pthread\_create(&tid, nullptr, threadFunction, nullptr);  while (true) {  // Simulate main thread activity  sleep(2);  pthread\_mutex\_lock(&mutex);  state = !state;  pthread\_cond\_signal(&cond);  pthread\_mutex\_unlock(&mutex);  }  pthread\_join(tid, nullptr);  return 0;  } |

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

| **Principles(s):** Embrace a Secure Coding Standard – Properly closing files can prevent information leakage and conserve resources within the program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft | 2023.1 | CERT\_C-FIO42-a | Ensure resources are freed. |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Leak. |
| Astree | 22.10 |  | Supported, but no explicit checker. |
| Polyspace Bug Finder | R2023b | CERT C: Rule FIO42-C | Checks for resource leaks. |

### 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.

To streamline adherence to our standards, we'll establish a centralized policy repository, integral to our assessment and planning procedures. Here, we'll house all organizational policies, encompassing regulatory compliance, release prerequisites, and general operational guidelines. This repository serves as the foundation for automating and enforcing policies. During verification and testing, we'll leverage this repository, allowing automated tools to evaluate risks, issue alerts, and deliver notifications. Automating compliance enhances our efficiency and minimizes repetitive tasks. Furthermore, we can automate transition and health check phases, leveraging automated penetration testing to reduce false alarms and facilitate issue resolution.

Taking it a step further, we'll fully automate the remainder of our production process. Implementing a logging system to store logs in a database enables us to identify vulnerabilities and thwart potential attacks. We'll employ methods such as signature checks, ensure data integrity, and implement multiple security layers to fortify our system. To maintain system resilience, automatic save points will be established, serving as clean backups. In the event of an attack or system malfunction, these save points allow us to revert to a secure version.

### 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | High | Probable | High | P6 | 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 | High | Probable | High | P6 | L2 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-010-CPP | Medium | Unlikely | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Protecting stored data on devices is called encryption at rest. Whether it's on hard drives or in databases, if it's shielded with encryption in those states, it all falls under the same category. Including this encryption method in the policy helps shield data in case of a breach. Hackers might get their hands on the data, but they won't be able to see what it is because of the encryption. Essentially, the data becomes unreadable and worthless to the hacker. Without encryption at rest, data is vulnerable to exploitation in the event of a breach. |
| Encryption in flight | Protecting data as it moves from one place to another is often referred to as encryption in transit. Data is always on the move, traveling between networks and users, or the other way around. While data is in transit, it could be intercepted, which is why encrypting it during this time is crucial. Encrypting data while it's in transit ensures that unauthorized individuals can't view it. The data gets encrypted before it's sent and decrypted once it reaches its destination. Implementing this policy helps minimize the harm that could result from a data breach. |
| Encryption in use | Protective measures can limit access to data in software applications to only the intended user. Data is vulnerable while it's actively used, but encrypting it in the computer's memory prevents attackers from spying on it. This policy is crucial for securing data and preventing exploitation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication involves confirming that the user trying to log into the system possesses the correct credentials to gain access. It restricts entry to the system according to the permissions granted by the admin. Systems may employ various authentication methods, like physical key cards, biometric features, or two-factor authentication. This policy is important for its security measures, which restrict access to authorized personnel. |
| Authorization | Authorization aims to restrict the actions users can take within the network. By limiting users' capabilities in the system, the risk of malware or vulnerabilities being introduced decreases. When implementing authorization, it's best to follow the principle of "less is best." This means that the less access users have, the better the prevention of attacks. |
| Accounting | Accounting involves maintaining a record of users' actions within the system. By documenting activities while individuals are on the network, administrators can access a history to review when suspicious behavior is identified. This approach can also notify administrators of any unauthorized attempts to access or modify data. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| Project 1 | 04/14/2024 | Project 1 | Tyler Turnbull |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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