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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | This validation acts as a critical line of defense, thwarting numerous potential threats, such as injection attacks, where malicious actors embed harmful data in input fields to manipulate the system. Effective input validation not only checks for correctness and relevance but also sanitizes data to remove any unintended or hazardous elements. By rigorously validating input, systems can prevent the processing of harmful data that could compromise security, maintain data integrity, and ensure the smooth operation of applications. |
| 1. Heed Compiler Warnings | Addressing compiler warnings is a proactive measure to enhance code quality and security. By resolving these warnings, developers can prevent potential security breaches and bugs that might not be evident during initial testing but could cause significant problems later. Ignoring compiler warnings is akin to overlooking potential weaknesses in the system that attackers could exploit. Treating compiler warnings with the same seriousness as errors is a fundamental practice in developing robust, secure, and reliable software. |
| 1. Architect and Design for Security Policies | This principle advocates for a proactive approach to security, where systems are designed with the anticipation of potential threats and are equipped with mechanisms to enforce security policies effectively. It requires a thorough understanding of the system's use cases, the data it will handle, and the potential security risks associated with its environment. This foundational focus on security helps in establishing a robust framework that supports secure operations throughout the system's lifecycle. |
| 1. Keep It Simple | By adopting simplicity, organizations can enhance the reliability and security of their systems. This approach involves minimizing the number of components, reducing the complexity of interactions, and avoiding unnecessary features that could introduce vulnerabilities. When systems are simple, it's easier for developers and security professionals to conduct thorough reviews. Keeping systems simple helps in creating a more predictable and controlled environment, where security can be more effectively managed |
| 1. Default Deny | Also known as "deny by default" or "default deny posture," is a fundamental security concept where access permissions are not granted unless explicitly allowed. This means that, by default, access to resources is denied, and only specific, authorized permissions are granted based on a need-to-access basis. |
| 1. Adhere to the Principle of Least Privilege | This dictates that individuals, systems, and programs should only have the minimum levels of access—or permissions—necessary to perform their tasks. This principle reduces the risk of unauthorized access or unintended actions that could compromise security. If a user's account is compromised, the principle of least privilege limits the actions that an attacker can perform, thereby containing the breach and reducing its impact. |
| 1. Sanitize Data Sent to Other Systems | This precautionary measure is crucial in preventing the inadvertent leakage of sensitive information and guarding against various forms of cyber threats. By sanitizing data, organizations can ensure that only clean, appropriate, and intended information is shared, thereby maintaining the integrity and confidentiality of the data while mitigating the risk of malicious exploitation. This principle is fundamental in developing secure coding practices and maintaining robust data exchange protocols in an interconnected digital ecosystem. |
| 1. Practice Defense in Depth | Refers to a multi-layered strategy in cybersecurity, where multiple security measures are employed to protect the information technology infrastructure of an organization. This approach is akin to having a series of defensive walls, where if an attacker breaches one layer, they are immediately faced with another. It's about not relying on a single security measure but instead having multiple safeguards at various levels – physical, technical, and administrative. |
| 1. Use Effective Quality Assurance Techniques | This involves a systematic process of checking to see whether a product or service being developed is meeting specified requirements. Peer reviews or code reviews involve scrutinizing code changes by team members before they are merged into the main codebase. This helps in catching defects early and fosters knowledge sharing. Automated code analysis tools can also help identify potential issues. |
| 1. Adopt a Secure Coding Standard | This approach involves following a set of guidelines and best practices that reduce vulnerabilities and ensure the integrity of available data. Select secure coding standards that are relevant to your programming languages and technologies. Standards such as OWASP's Secure Coding Practices, CERT Secure Coding Standards, or industry-specific guidelines provide a good foundation. |

### 

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not define a C-style variadic function |

| **Noncompliant Code** |
| --- |
| Using C-style variadic functions allows for an arbitrary number of parameters of varying types, which compromises type safety and can lead to undefined behavior if the function expects a type different from what is provided. |
| #include <iostream>  #include <cstdarg>  void printNumbers(int count, ...) {  va\_list args;  va\_start(args, count);  for (int i = 0; i < count; ++i) {  int number = va\_arg(args, int);  std::cout << number << ' ';  }  va\_end(args);  }  int main() {  printNumbers(3, 1, 2, 3);  return 0;  } |

| **Compliant Code** |
| --- |
| Using std::vector or variadic templates ensures type safety and allows the function to handle variable numbers of parameters without sacrificing the benefits of strong typing. |
| #include <iostream>  #include <vector>  void printNumbers(const std::vector<int>& numbers) {  for (int number : numbers) {  std::cout << number << ' ';  }  }  int main() {  printNumbers({1, 2, 3});  return 0;  } |

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

| **Keep It Simple -** C-style variadic functions, such as those using stdarg.h in C (e.g., printf, scanf), can complicate the implementation because they allow for an arbitrary number of arguments. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | **function-ellipsis** | Fully checked |
| Clang | 3.9 | cert-dcl50-cpp | Checked by clang-tidy. |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-DCL50-a** | Functions shall not be defined with a variable number of arguments |
| RuleChecker | 22.10 | **function-ellipsis** | Fully checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not cast to an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| Directly casting an integer to an enumeration type without checking if the value is within the enumeration's defined range can lead to undefined behavior or logical errors. |
| enum Color { RED, GREEN, BLUE };  int main() {  int num = 10;  Color color = static\_cast<Color>(num); // Potentially out-of-range  return 0;  } |

| **Compliant Code** |
| --- |
| Including a range check before casting ensures that the integer value is valid for the enumeration, preventing undefined behavior and increasing the code's robustness. |
| enum Color { RED, GREEN, BLUE };  Color intToColor(int num) {  if (num >= RED && num <= BLUE) {  return static\_cast<Color>(num);  }  // Handle error or default case  return RED;  }  int main() {  Color color = intToColor(1);  return 0;  } |

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

| **Default Deny -** When you avoid casting to out-of-range values of an enumeration, you are effectively applying a "default deny" approach to data types and values in your program.  **Adhere to the Principle of Least Privilege -** Casting to an out-of-range enumeration value in programming can lead to undefined behavior or logic errors, where the system might operate under conditions it was not designed to handle. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |
| Parasoft C/C+  +test | 2023.1 | **CERT\_CPP-INT50-a** | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| RuleChecker | 22.10 | **cast-integer-to-enum** | Partially checked |
| Polyspace Bug Finder | R2023b | CERT C++: INT50-  CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| Allocating insufficient space for a string plus its null terminator leads to buffer overflow, which can cause program crashes and security vulnerabilities. |
| #include <iostream>    void f() {    char buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Allocating sufficient space for all characters and the null terminator avoids buffer overflow, ensuring the program's stability and security. |
| #include <iostream>  #include <string>    void f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Validate Input Data -** By guaranteeing sufficient space for strings, including the null terminator, you are essentially validating that the storage allocated for string data is adequate to handle the input safely.  **Practice Defense in Depth -** Ensuring that every string storage allocation is properly sized adds a layer of security by mitigating risks associated with memory management errors, such as buffer overflows.  **Use Effective Quality Assurance Techniques -** Ensuring that strings are properly terminated and fit within their allocated space is a basic yet critical check that should be included in unit tests, integration tests, and code reviews. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likley | Meduim | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO** **LANG.MEM.TO** | No space for null terminator  Buffer overrun Type overrun |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Polyspace Bug Finder | R2023b | CERT C++: STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| RuleChecker | 22.10 | **stream-input-char-array** | Partially checked |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| Dereferencing a pointer after the memory it points to has been freed can lead to undefined behavior, as the memory might have been reallocated and modified. |
| #include <iostream>  int main() {  int\* ptr = new int(10);  delete ptr;  std::cout << \*ptr; // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| Setting a pointer to nullptr after freeing the memory prevents dereferencing freed memory, eliminating the risk of accessing invalid memory locations. |
| #include <iostream>  int main() {  int\* ptr = new int(10);  delete ptr;  ptr = nullptr; // Set pointer to nullptr after deletion  // Now ptr is not dereferenced after deletion  return 0;  } |

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

| **Heed Compiler Warnings -** Paying attention to and addressing compiler warnings, developers can prevent potential vulnerabilities like accessing freed memory, which can lead to unpredictable behavior and security risks such as crashes and exploitable conditions.  **Practice Defense in Depth -** Applying defense in depth to memory management means incorporating checks and balances such as runtime bounds checking.  **Use Effective Quality Assurance Techniques -** Thorough testing and validation procedures helps catch issues such as accessing freed memory before the software is released. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.1 | **C++4303, C++4304** | - |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| Coverity | V7.5.0 | |  |  | | --- | --- | |  | USE\_AFTER\_FREE | | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| PVS-Studio | 7.30 | V586, V774 | - |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| Using std::exit or similar functions within catch blocks to terminate a program can bypass destructor calls and other cleanup operations, potentially leading to resource leaks and making it harder to diagnose the exception's cause. |
| #include <iostream>  #include <cstdlib>  int main() {  try {  // Code that might throw an exception  throw std::runtime\_error("An error occurred");  } catch (...) {  std::cerr << "Exception caught. Exiting.";  std::exit(1);  }  return 0;  } |

| **Compliant Code** |
| --- |
| Handling exceptions without abruptly terminating the program allows for appropriate resource cleanup and error logging, facilitating debugging and ensuring resource integrity. |
| #include <iostream>  int main() {  try {  // Code that might throw an exception  throw std::runtime\_error("An error occurred");  } catch (const std::exception& e) {  std::cerr << "Exception caught: " << e.what() << '\n';  // Handle the exception without terminating abruptly  }  return 0;  } |

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

| **Architect and Design for Security Policies -** Designing with robust error handling and recovery mechanisms in place aligns with creating a secure environment that anticipates failures.  **Keep It Simple -** A simpler design often results in more predictable behavior and easier maintenance, which can contribute to smoother error handling.  **Use Effective Quality Assurance Techniques -** Regular stress testing, scenario testing, and fault injection can help developers understand how the application behaves. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stdib-use | Partially checked |
| CodeSonar | 8.1p0 | **BADFUNC.ABORT BADFUNC.EXIT** | Use of abort Use of exit |
| Helix QAC | 2024.1 | **C++5014** | - |
| Klocwork | 2024.1 | **MISRA.TERMINATE CERT.ERR.ABRUPT\_TERM** | - |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Prevent Reading Uninitialized Memory |

| **Noncompliant Code** |
| --- |
| Attempts to read an uninitialized variable, leading to undefined behavior. |
| #include <iostream>  int main() {  int uninitializedVariable;  std::cout << "Uninitialized value: " << uninitializedVariable << std::endl; // Non-compliant: reading uninitialized memory  return 0;  } |

| **Compliant Code** |
| --- |
| Initializes the variable before use, ensuring that no undefined or unpredictable values are read. |
| #include <iostream>  int main() {  int initializedVariable = 0; // The variable is initialized  std::cout << "Initialized value: " << initializedVariable << std::endl;  return 0;  } |

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

| **Validate Input Data -** Initializing memory before it's read, you effectively validate that the memory's state is safe and predictable.  **Heed Compiler Warnings -** Paying attention to these warnings and addressing them can prevent many issues related to uninitialized memory.  **Practice Defense in Depth -** Implementing multiple layers of checks and safeguards against using uninitialized memory forms part of a defense-in-depth strategy.  **Use Effective Quality Assurance Techniques -** QA processes ensure that software is tested under conditions that might reveal the use of uninitialized memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **uninitialized-read** | Partially checked |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| CodeSonar | 8.1p0 | **LANG.STRUCT.RPL LANG.MEM.UVAR** | Return pointer to local Uninitialized variable |
| Helix QAC | [Insert text.] | **DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978** | - |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Ensure Exception Safety |

| **Noncompliant Code** |
| --- |
| Not managing resources properly in the face of exceptions can lead to resource leaks, inconsistent state, or undefined behavior. |
| #include <iostream>  #include <stdexcept>  class Resource {  public:  Resource() { std::cout << "Resource Acquired\n"; }  ~Resource() { std::cout << "Resource Released\n"; }  };  void riskyOperation() {  Resource res;  throw std::runtime\_error("Failed Operation");  // The destructor for res is not called, leading to a potential resource leak.  }  int main() {  try {  riskyOperation();  } catch (const std::exception& e) {  std::cerr << "Exception caught: " << e.what() << '\n';  }  return 0;  } |

| **Compliant Code** |
| --- |
| Using RAII (Resource Acquisition Is Initialization) ensures that resources are properly released even when an exception is thrown. |
| #include <iostream>  #include <stdexcept>  #include <memory>  class Resource {  public:  Resource() { std::cout << "Resource Acquired\n"; }  ~Resource() { std::cout << "Resource Released\n"; }  };  void riskyOperation() {  std::unique\_ptr<Resource> res = std::make\_unique<Resource>();  throw std::runtime\_error("Failed Operation");  // res's destructor is called, releasing the resource even if an exception is thrown.  }  int main() {  try {  riskyOperation();  } catch (const std::exception& e) {  std::cerr << "Exception caught: " << e.what() << '\n';  }  return 0;  } |

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

| **Architect and Design for Security Policies -** Ensuring exception safety involves creating systems that are robust against exceptional conditions.  **Keep It Simple -** A simpler system with straightforward logic is easier to verify for exception safety.  **Use Effective Quality Assurance Techniques - T**echniques such as unit testing help verify that the software behaves as expected under exceptional circumstances and that it recovers gracefully from errors.  **Adopt a Secure Coding Standard -** These standards often include best practices for managing exceptions, such as ensuring that exceptions do not lead to resource leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2024.1 | **C++4075, C++4076** | - |
| LDRA tool suite | 9.7.1 | **527 S, 56 D, 71 D** | Partially implemented |
| Parasoft C/C++ test | 2023.1 | **CERT\_CPP-ERR56-a CERT\_CPP-ERR56-b** | Always catch exceptions Do not leave 'catch' blocks empty |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Management | [STD-008-CPP] | Dynamically allocated resources must be properly deallocated. |

| **Noncompliant Code** |
| --- |
| Failure to deallocate dynamically allocated memory leads to memory leaks, which consume unnecessary memory and can cause application or system instability. |
| #include <iostream>  void createLeak() {  int\* ptr = new int(42);  // No delete call, ptr is not deallocated  }  int main() {  createLeak();  std::cout << "Memory leak created." << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Ensure that every new is matched with a delete to prevent memory leaks and ensure the stability of the application. |
| #include <iostream>  void allocateAndDeallocate() {  int\* ptr = new int(42);  delete ptr; // Properly deallocate memory  }  int main() {  allocateAndDeallocate();  std::cout << "No memory leaks!" << std::endl;  return 0;  } |

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

| **Keep It Simple - S**imple designs reduce the complexity of resource management and make it easier to follow and implement proper cleanup patterns.  **Adhere to the Principle of Least Privilege -** Managing resources tightly and minimizing their lifetime aligns with limiting the "privileges" of these resources to only what is necessary for their intended use.  **Practice Defense in Depth -** Applying multiple layers of checks and controls around resource allocation and deallocation can help prevent and mitigate the effects of resource leaks or improper deallocation.  **Use Effective Quality Assurance Techniques -** Memory and resource leak detection tools can be part of QA testing to catch issues where dynamically allocated resources are not properly deallocated.  **Adopt a Secure Coding Standard -** Following a secure coding standard can provide guidelines and best practices for managing dynamically allocated resources. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | **invalid\_dynamic\_memory\_allocation dangling\_pointer\_use** | - |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-MEM51** | - |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 8.1p0 | **ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | [STD-009-CPP] | Not closing files can lead to resource leaks. |

| **Noncompliant Code** |
| --- |
| Failing to close a file after its use can result in resource leaks and incomplete data writing due to unflushed buffers. |
| #include <fstream>  void writeToFile() {  std::ofstream file("example.txt");  file << "Hello, file!";  // File is not explicitly closed.  }  int main() {  writeToFile();  return 0;  } |

| **Compliant Code** |
| --- |
| Explicitly closing files or using RAII to manage file resources ensures that resources are released and data is properly written. |
| #include <fstream>  void writeToFile() {  std::ofstream file("example.txt");  file << "Hello, file!";  file.close(); // Explicitly close the file  }  int main() {  writeToFile();  return 0;  } |

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

| **Default Deny -** This principle ensures that the opening of file streams is always paired with their closure, thereby preventing resource leaks.  **Adhere to the Principle of Least Privilege -** In terms of file management, it means that file handles should be opened only when needed and closed immediately after their use is completed.  **Practice Defense in Depth -** This principle can be applied to resource management by setting up multiple safeguards.  **Use Effective Quality Assurance Techniques -** Comprehensive testing strategies to ensure that all parts of a program function as intended, including the proper management of file handles.  **Adopt a Secure Coding Standard -** Helps prevent common mistakes such as not closing files, thereby avoiding potential resource leaks and other associated issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2024.1 | **RH.LEAK** | [Insert text.] |
| Parasoft Insure++ | - | - | Runtime detection |
| Helix QAC | 2024.1 | |  |  | | --- | --- | | DF4786, DF4787, DF4788 |  | | - |
| Polyspace Bug Finder | R2023b | |  |  | | --- | --- | | CERT C++: FIO51-CPP |  | | Checks for resource leak (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object-Oriented Programming | [STD-010-CPP] | Use Virtual Destructors for Polymorphic Classes |

| **Noncompliant Code** |
| --- |
| Deleting a derived class object through a base class pointer without a virtual destructor does not call the derived class's destructor, leading to potential resource leaks or undefined behavior. |
| #include <iostream>  class Base {  public:  ~Base() { std::cout << "Base destructor\n"; }  };  class Derived : public Base {  public:  ~Derived() { std::cout << "Derived destructor\n"; }  };  int main() {  Base\* obj = new Derived();  delete obj; // Only calls Base's destructor, not Derived's  return 0;  } |

| **Compliant Code** |
| --- |
| A virtual destructor in the base class ensures that the correct destructors are called, even when deleting through a base class pointer. |
| #include <iostream>  class Base {  public:  virtual ~Base() { std::cout << "Base destructor\n"; }  };  class Derived : public Base {  public:  ~Derived() override { std::cout << "Derived destructor\n"; }  };  int main() {  Base\* obj = new Derived();  delete obj; // Calls Derived's destructor followed by Base's destructor  return 0;  } |

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

| **Architect and Design for Security Policies -** This design choice prevents resource leaks and undefined behavior, which are critical aspects of secure and reliable software architecture.  **Keep It Simple -** Ensures that the destructor for the most derived class is called, even if objects are deleted through pointers to base classes, making the system's behavior more predictable and easier to understand.  **Practice Defense in Depth - I**mplementing virtual destructors as part of a defense in depth strategy can prevent subtle bugs that may lead to resource leaks or improper resource cleanup.  **Use Effective Quality Assurance Techniques -** Tools like static analyzers can automatically detect when virtual destructors are missing in polymorphic classes.  **Adopt a Secure Coding Standard -** helps to avoid common pitfalls in C++ programming that can lead to improper behavior when polymorphic objects are deleted. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-OOP52** | - |
| Clang | 3.9 | -Wdelete-non-virtual-dtor | - |
| RuleChecker | 22.10 | **non-virtual-public-destructor-in-non-final-class** | Partially checked |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-OOP52-a** | Define a virtual destructor in classes used as base classes which have virtual functions |

### 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 | Probable | Medium | High | 1 |
| STD-002-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Probably | Medium | High | 1 |
| STD-005-CPP | Low | Probably | Medium | Low | 3 |
| STD-006-CPP | High | Probably | Medium | Medium | 1 |
| STD-007-CPP | High | Likely | High | Medium | 1 |
| STD-008-CPP | High | Likely | Medium | High | 1 |
| STD-009-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-010CPP | Low | Likely | Low | Medium | 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 at rest | Encryption at rest involves securing data by encrypting it while it is stored on a hard drive or any other form of permanent storage. Encryption should be implemented at the disk, database, or application level depending on the sensitivity of the data and compliance requirements. The keys used for encryption must be managed securely, involving key rotation and access restrictions. Encryption at rest protects against data theft from stolen physical devices or unauthorized access to storage systems. |
| Encryption in flight | Refers to the protection of data while it is being transmitted across networks. Implementation involves configuring network and application-level settings to enforce secure connections. Encryption in flight prevents attackers from intercepting, viewing, or altering data during transmission. |
| Encryption in use | Unlike data at rest or in flight, this protects the integrity and confidentiality of data during computation. Applications should be designed to limit exposure of sensitive data in memory and only decrypt data at the point of use. Encryption in use shields sensitive data from threats that arise during processing, such as unauthorized access by system administrators, compromises of the operating system. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of verifying the identity of a user, device, or entity before granting access to systems and data. It should be enforced at all system and application access points to ensure that only verified users can gain entry. Authentication prevents unauthorized access by ensuring that only legitimate users can access the system. It is a critical first line of defense against intrusions and helps in maintaining the overall security posture of the organization. |
| Authorization | The process of granting or denying specific permissions to an authenticated user, based on their role or individual permissions. Users should have only the minimum necessary permissions to perform their tasks. Authorization ensures that users can only perform actions within their scope of responsibilities, preventing misuse of permissions that could lead to data breaches or other security incidents. |
| Accounting | Refers to tracking and recording user activities within the system. Logs should be protected from tampering, and tools should be employed to analyze these records for signs of suspicious activity. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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