**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 | Validation data is important to prevent security vulnerabilities, such as cross-site scripting, SQL Injection, or buffer overflow attacks. What is means to validate is to check for data for type, length, format, and range to confirm it meets the predefined criteria. Catching harmful input early in the process. Implementing robust input checks can significantly reduce the risk of malicious exploits. |
| 1. Heed Compiler Warnings | A complier warning can indicate potential flaws in the code that could lead to security vulnerabilities. Warning can result in software that is more susceptible to attacks. By addressing and resolving all complier warnings, this will ensure robust and secure code, also identifying and mitigating issues early in the development cycle. |
| 1. Architect and Design for Security Policies | In all great system designs, security is an integral part of the architecture and design. Implementing security policies such as access controls, authentication, and encryption from the start. Keeping security in the for front, developers can create systems that are inherently more secure and resilient to threats. This proactive approach helps in minimizing vulnerabilities and ensuring compliance with security standards. |
| 1. Keep It Simple | By keeping the system simple, it helps reduce the risk of introducing security vulnerabilities. Complex systems are more challenging to understand, test, maintain, and patch, increasing the likelihood of errors and overlooked weaknesses. By keeping systems simple, developers can more easily identify and mitigate potential security issues. |
| 1. Default Deny | By design, deny principle involves configuring systems to deny access first, only allowing access when explicitly granted. By implementing default deny policies, organizations can better control and monitor access to sensitive information and systems. This approach strengthens security by making unauthorized access attempts more visible and manageable. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege dictates that users and processes should have minimum level of access necessary to perform their functions. This can help limit the potential damage caused by compromised accounts or malicious insiders. By adhering to this principle, organizations can reduce the risk of unauthorized access and data breaches. Implementing strict access controls and regularly reviewing privileges helps in maintaining a secure environment. |
| 1. Sanitize Data Sent to Other Systems | All data needs to be sanitized when sent to an external system to prevent injection attacks and data leakage. This involves removing or encoding special characters, ensuring that data adheres to expected formats and validating data integrity. This will ensure the protection of both sending and receiving systems from potential threats. By ensuring that data is clean and safe, organizations can prevent many common security issues. |
| 1. Practice Defense in Depth | Defense in depth is a layered security approach that involves implementing multiple security measures at different levels. This strategy ensures that if one layer is breached, additional layers of defense can still provide protection. All the while, no one security feature can ensure the safety of an entire system, therefore having multiple layers protects against a wide range of threats. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques such as code reviews, automated testing, and static analysis help in identifying and mitigating security vulnerabilities. Effective QA practices ensure that software meets security standards before deployment. By integrating security checks into the QA process, organizations can catch issues early improve and overall security posture of their software. |
| 1. Adopt a Secure Coding Standard | Secure coding standards provide guidelines for writing code that is resistant to security vulnerabilities. Adopting these standards helps developers consistently implement security best practices across their projects. Regular training and updates to coding standards are necessary to keep up with evolving threats. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Qualifying a reference type with const or volatile can lead to confusing and undefined behavior, as it suggests that the reference itself is immutable or volatile rather than the object it references. This practice can cause misunderstandings in the codebase, leading to maintenance challenges and potential security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code snippet incorrectly qualifies a reference type with const, which does not convey the intended immutability of the referenced object. |
| #include <iostream>  void f(char c) {  const char &p = c;  p = 'p'; // Error: read-only variable is not assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the const qualifier. |
| #include <iostream>  void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):** Keep it Simple - This principle applies because avoiding unnecessary qualifiers like const or volatile on reference types keeps the code straightforward, reducing the chance of misunderstandings and maintaining simplicity in the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 |  |
| Helix QAC | 2024.2 | C++0014 |  |
| Klocwork | 2024.2 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |
| Polyspace Bug Finder | R2024a | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| Clang | 3.9 |  | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options. |
| SonarQube C/C++ Plugin | 4.10 | S3708 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Never hard code sensitive information** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Hardcoding sensitive information such as passwords, cryptographic keys, or personal data directly in the source code exposes it to potential attackers who can easily extract it from the binary or source files. This practice compromises security and can lead to unauthorized access and data breaches. Ensuring that sensitive information is stored securely and accessed safely helps maintain the confidentiality and integrity of the data. |

| **Noncompliant Code** |
| --- |
| This code snippet demonstrates hardcoding a password directly in the source code, which is a security risk. |
| /\* Returns nonzero if authenticated \*/  int authenticate(const char\* code);    int main() {  if (!authenticate("correct code")) {  printf("Authentication error\n");  return -1;  }    printf("Authentication successful\n");  // ...Work with system...  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code retrieves the password from a secure location, such as an environment variable, ensuring that sensitive information is not hardcoded in the source code. |
| /\* Returns nonzero if authenticated \*/  int authenticate(const char\* code);    int main() {  #define CODE\_LEN 50  char code[CODE\_LEN];  printf("Please enter your authentication code:\n");  fgets(code, sizeof(code), stdin);  int flag = authenticate(code);  memset\_s(code, sizeof(code), 0, sizeof(code));  if (!flag) {  printf("Access denied\n");  return -1;  }  printf("Access granted\n");  // ...Work with system...  return 0;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard - Hardcoding sensitive information violates secure coding practices. Following this principle ensures that sensitive data is handled securely, reducing the risk of unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | HARDCODED.AUTH  HARDCODED.DNS  HARDCODED.KEY  HARDCODED.SALT  HARDCODED.SEED | Hardcoded Authentication  Hardcoded DNS Name  Hardcoded Crypto Key  Hardcoded Crypto Salt  Hardcoded Seed in PRNG |
| Helix QAC | 2024.2 | C3122  C++3842 |  |
| Klocwork | 2024.2 | HCC  HCC.PWD  HCC.USER  CXX.SV.PWD.PLAIN  CXX.SV.PWD.PLAIN.LENGTH  CXX.SV.PWD.PLAIN.ZERO |  |
| Parasoft C/C++test | 2023.1 | CERT\_C-MSC41-a | Do not hard code string literals |
| PC-lint Plus | 1.4 | 2460 | Assistance provided: reports when a literal is provided as an argument to a function parameter with the ‘noliteral’ argument Semantic; several Windows API functions are marked as such and the ‘-sem’ option can apply it to other functions as appropriate |
| Polyspace Bug Finder | R2024a | CERT C: Rule MSC41-C | Checks for hard coded sensitive data (rule partially covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Attempting to create a std::string from a null pointer can lead to undefined behavior, including crashes or security vulnerabilities. The std::string constructor expects a valid pointer to a null-terminated character array, and passing a null pointer violates this expectation. Ensuring that pointers are valid before creating std::string objects prevents these potential issues. |

| **Noncompliant Code** |
| --- |
| This code snippet attempts to create a std::string from a null pointer, which can result in undefined behavior. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** Validate Input Data - This principle is crucial here because ensuring that pointers are not null before creating a std::string object prevents undefined behavior and potential security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2024.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2024.2 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2024a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Do not specify the bound of a character array initialized with a string literal** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Specifying the bound of a character array when initializing it with a string literal can lead to errors if the bound is incorrect, potentially causing buffer overflows or truncation of the string. This can introduce vulnerabilities, including SQL injection, if user input is not handled properly. Ensuring that character arrays are correctly sized automatically to the length of the string literal prevents these issues and promotes safer code. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example initializes an array of characters using a string literal that defines one character more (counting the terminating '\0') than the array can hold: |
| const char s[3] = "abc"; |

| **Compliant Code** |
| --- |
| This compliant solution does not specify the bound of a character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character. |
| const char s[] = "abc"; |

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

| **Principles(s):** Architect and Design for Security Policies - This principle is relevant because designing code that automatically handles string lengths reduces the risk of buffer overflows, a common security vulnerability, thus embedding security into the design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported: Astrée can detect subsequent code defects that this rule aims to prevent. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR11 |  |
| Compass/ROSE |  |  |  |
| ECLAIR | 1.2 | CC2.STR36 | Fully implemented |
| Helix QAC | 2024.2 | C1312 |  |
| LDRA tool suite | 9.7.1 | 404 S | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR11-a | Do not specify the bound of a character array initialized with a string literal |
| PC-lint Plus | 1.4 | 784 | Partially supported |
| Polyspace Bug Finder | R2024a | CERT C: Rec. STR11-C | Checks for missing null in string array (rec. partially covered) |
| Splint | 3.1.1 |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Accessing freed memory can lead to undefined behavior, including crashes, data corruption, and security vulnerabilities such as use-after-free attacks. When memory is freed, the data it contained is no longer valid, and accessing it can compromise program stability and security. Ensuring that pointers are properly managed and set to null after freeing memory prevents inadvertent access to invalid memory locations. |

| **Noncompliant Code** |
| --- |
| This code snippet demonstrates accessing memory after it has been freed, which leads to undefined behavior and potential security risks. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** Practice Defense in Depth - Accessing freed memory can lead to severe vulnerabilities, including use-after-free attacks. This principle applies because ensuring memory is managed correctly adds a layer of defense against such attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| 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. |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Choose an appropriate termination strategy** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Choosing an appropriate termination strategy is essential for handling unexpected conditions and errors in a controlled manner. Using assertions helps catch programming errors during development by terminating the program if a condition is not met, ensuring that issues are identified and resolved early. However, assertions should be used judiciously and not for runtime error handling, as they do not provide a mechanism for recovery or graceful degradation. |

| **Noncompliant Code** |
| --- |
| This code snippet uses assertions inappropriately for handling runtime errors, which may not be enabled in release builds, leading to undetected errors in production. |
| #include <stdlib.h>  #include <stdio.h>    int write\_data(void) {  const char \*filename = "hello.txt";  FILE \*f = fopen(filename, "w");  if (f == NULL) {  /\* Handle error \*/  }  fprintf(f, "Hello, World\n");  /\* ... \*/  abort(); /\* Oops! Data might not be written! \*/  /\* ... \*/  return 0;  }    int main(void) {  write\_data();  return EXIT\_SUCCESS;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to abort() is replaced with exit(), which guarantees that buffered I/O data is flushed to the file descriptor and the file descriptor is properly closed: |
| #include <stdlib.h>  #include <stdio.h>    int write\_data(void) {  const char \*filename = "hello.txt";  FILE \*f = fopen(filename, "w");  if (f == NULL) {  /\* Handle error \*/  }  fprintf(f, "Hello, World\n");  /\* ... \*/  exit(EXIT\_FAILURE); /\* Writes data and closes f \*/  /\* ... \*/  return 0;  }    int main(void) {  write\_data();  return EXIT\_SUCCESS;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques - Selecting the right termination strategy, like replacing abort() with exit(), ensures that programs fail gracefully, improving overall software quality and reducing the risk of unhandled errors in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_C-ERR04-a  CERT\_C-ERR04-b  CERT\_C-ERR04-c | The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  The 'exit()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |
| PC-lint Plus | 1.4 | 586 | Fully supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Honor exception specifications** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Honoring exception specifications ensures that functions only throw exceptions that are declared, promoting code reliability and predictability. This practice helps in maintaining a clear contract between the function and its callers, reducing the likelihood of unexpected exceptions and simplifying error handling. Adhering to exception specifications enhances code maintainability and robustness, making it easier to debug and manage exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) noexcept(**true**) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) {  v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):** Adopt a Secure Coding Standard - Adhering to exception specifications ensures predictable behavior in the code, which is a fundamental aspect of secure coding standards. It promotes reliability and maintainability in error handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 56 D | Partially implemented |
| Parasoft C/C++Test | 2023.1 | CERT\_CPP-ERR55-a | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |
| Polyspace Bug Finder | R2024a | CERT C++: ERR55-CPP | Checks for noexcept functions exiting with exception (rule fully covered) |
| RuleChecker | 22.10 | unhandled-throw-noexcept | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not slice derived objects** |
| --- | --- | --- |
| Object Oriented Programming | STD-008-CPP | Object slicing occurs when a derived class object is assigned to a base class object, resulting in the loss of the derived part of the object. This can lead to undefined behavior and loss of important data or functionality. Ensuring that derived objects are not sliced preserves the integrity of the objects and maintains the polymorphic behavior expected in object-oriented programming. Proper use of references or pointers to base class types prevents slicing and ensures that derived class features are accessible. |

| **Noncompliant Code** |
| --- |
| This code snippet demonstrates object slicing by assigning a derived class object to a base class object, causing loss of the derived part of the object. |
| class Base {  public:  virtual void display() const {  std::cout << "Base display" << std::endl;  }  };  class Derived : public Base {  public:  void display() const override {  std::cout << "Derived display" << std::endl;  }  };  void showDisplay(Base b) {  b.display();  }  int main() {  Derived d;  showDisplay(d); // Object slicing occurs  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code uses references to prevent object slicing, ensuring that the derived class object retains its full functionality when passed to the function. |
| class Base {  public:  virtual void display() const {  std::cout << "Base display" << std::endl;  }  };  class Derived : public Base {  public:  void display() const override {  std::cout << "Derived display" << std::endl;  }  };  void showDisplay(const Base& b) {  b.display();  }  int main() {  Derived d;  showDisplay(d); // No object slicing  return 0;  } |

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

| **Principles(s):** Keep it Simple - avoiding object slicing is crucial to preserving the polymorphic behavior expected in OOP. By ensuring derived objects are not sliced, the integrity of the object's data and behavior is maintained, supporting the key OOP principle of inheritance and polymorphism. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-OOP51-a | Avoid slicing function arguments / return value |
| Polyspace Bug Finder | R2024a | CERT C++: OOP51-CPP | Checks for object slicing (rule partially covered) |
| PVS-Studio | 7.32 | V1054 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Wrap functions that can spuriously wake up in a loop** |
| --- | --- | --- |
| Concurrency | STD-009-CPP | Functions that wait for conditions, such as std::condition\_variable::wait, can sometimes wake up without the condition being met due to spurious wakeups. To ensure the program behaves correctly, it is essential to wrap such functions in a loop that rechecks the condition after waking up. This practice ensures that the program only proceeds when the intended condition is genuinely satisfied, maintaining synchronization correctness and preventing subtle concurrency bugs. |

| **Noncompliant Code** |
| --- |
| This code monitors a linked list, using wait() to pause a thread until notified. The thread may resume due to spurious notifications, like from notify\_all(), even if the list is empty. By nesting wait() inside an if block, it fails to recheck the condition after notification, risking premature thread wakeup. |
| #include <condition\_variable>  #include <mutex>    struct Node {  void \*node;  struct Node \*next;  };    static Node list;  static std::mutex m;  static std::condition\_variable condition;    void consume\_list\_element(std::condition\_variable &condition) {  std::unique\_lock<std::mutex> lk(m);    if (list.next == nullptr) {  condition.wait(lk);  }    // Proceed when condition holds.  } |

| **Compliant Code** |
| --- |
| This compliant solution calls the wait() member function from within a while loop to check the condition both before and after the call to wait(). |
| #include <condition\_variable>  #include <mutex>    struct Node {  void \*node;  struct Node \*next;  };    static Node list;  static std::mutex m;  static std::condition\_variable condition;    void consume\_list\_element() {  std::unique\_lock<std::mutex> lk(m);    while (list.next == nullptr) {  condition.wait(lk);  }    // Proceed when condition holds.  } |

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

| **Principles(s):** Practice Defense in Depth - Wrapping such functions in a loop ensures that the program behaves correctly even in the case of spurious wakeups, adding an extra layer of reliability and protection against concurrency issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 | C++5019 |  |
| Klocwork | 2024.2 | CERT.CONC.WAKE\_IN\_LOOP |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CON54-a | Wrap functions that can spuriously wake up in a loop |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not copy a FILE object** |
| --- | --- | --- |
| Input Output | STD-010-CPP | Copying a FILE object can lead to undefined behavior and resource management issues, such as multiple file handles pointing to the same file stream. This can result in data corruption, crashes, and other unpredictable behaviors. Ensuring that FILE objects are not copied preserves the integrity of file operations and maintains proper resource management, preventing such issues. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can fail because a by-value copy of stdout is being used in the call to fputs(): |
| #include <stdio.h>    int main(void) {  FILE my\_stdout = \*stdout;  if (fputs("Hello, World!\n", &my\_stdout) == EOF) {  /\* Handle error \*/  }  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a copy of the stdout pointer to the FILE object is used in the call to fputs(): |
| #include <stdio.h>    int main(void) {  FILE \*my\_stdout = stdout;  if (fputs("Hello, World!\n", my\_stdout) == EOF) {  /\* Handle error \*/  }  return 0;  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems - By ensuring that FILE objects are not copied, the code avoids potential data corruption and undefined behavior, thereby maintaining the integrity and reliability of data operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 591 S | Fully implemented |
| PC-lint Plus | 1.4 | 9047 | Partially supported: reports when a FILE pointer is dereferenced |
| Polyspace Bug Finder | R2024a | CERT C: Rule FIO38-C | Checks for misuse of a FILE object (rule fully covered) |
| RuleChecker | 24.04 | file-dereference | Partially checked |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation



To automate enforcement and compliance with the standards defined in the Green Pace security policy, the existing DevOps process can be enhanced by integrating security practices into every phase of the software development lifecycle, as depicted in the DevSecOps diagram. In the pre-production phase, automation should be embedded during the "Design" stage by incorporating security requirements (such as OWASP principles) into the design process, and during the "Build" stage by utilizing automated security testing tools. In the "Verify and Test" phase, continuous integration pipelines can be configured to automatically run security checks, code quality assessments, and vulnerability scans. As the process transitions to production, automated tools should be employed to monitor and detect threats, leveraging Security Information and Event Management (SIEM) systems and intrusion detection systems. Additionally, automation in the "Respond" and "Maintain and Stabilize" phases ensures that incidents are managed swiftly, with predefined automated responses to identified threats and continuous monitoring to maintain security compliance. This holistic integration of automation across the DevSecOps lifecycle ensures that security is a continuous, embedded part of the development process, enhancing overall security posture while maintaining agility.

### 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 | Low | Unlikely | Low | P3 | L3 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | Low | Probably | Low | P6 | L2 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Medium | Probably | High | P4 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | Low | Probably | Medium | P4 | L3 |
| STD-009-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-010-CPP | Low | Probable | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 protects data stored on disks or other media, ensuring it remains unreadable without proper decryption keys if physical storage is compromised. This policy applies to all sensitive data stored by Green Pace, safeguarding confidentiality, and integrity against unauthorized access to storage devices. |
| Encryption in flight | Encryption in flight secures data during transmission over networks, protecting it from interception or tampering. This policy applies to all sensitive data communications, both internal and external, ensuring data confidentiality and integrity through methods like TLS/SSL. |
| Encryption in use | Encryption in use protects data while it's being processed in memory or during computation. This policy applies to scenarios where sensitive data is actively used, particularly in cloud environments, ensuring data remains secure even during processing. |

| 1. **Triple-A Framework** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users or systems before granting access. This policy mandates strong, multi-factor authentication for all Green Pace systems, ensuring that only authorized users can access sensitive resources. It applies to all user logins, securing the initial access point to protect against unauthorized entry. |
| Authorization | Authorization controls what authenticated users can do within the system. This policy enforces the principle of least privilege, ensuring that users and systems have only the necessary access for their roles. It applies to setting and reviewing user levels of access, database changes, and the addition of new users, reducing the risk of unauthorized actions. |
| Accounting | Accounting tracks and logs critical activities within the system. This policy requires logging of user logins, changes to the database, addition of new users, user access levels, and files accessed by users. This ensures accountability, aids in detecting suspicious activities, and supports compliance with security standards. |

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.5 | 07/14/2024 | Coding Standard | Joseph Les |  |
| 2.0 | 08/09/2024 | Document Completed | Joseph Les |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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