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



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 1](#_Toc1922306235)

[Contents 1](#_Toc1796563198)

[Overview 2](#_Toc68360143)

[Purpose 3](#_Toc1272431037)

[Scope 3](#_Toc229715314)

[Module Three Milestone 3](#_Toc1694534255)

[Ten Core Security Principles 3](#_Toc1360680000)

[C/C++ Ten Coding Standards 4](#_Toc287081243)

[Coding Standard 1 4](#_Toc690024984)

[Coding Standard 2 6](#_Toc669658826)

[Coding Standard 3 8](#_Toc1646520622)

[Coding Standard 4 10](#_Toc242705923)

[Coding Standard 5 12](#_Toc1803568971)

[Coding Standard 6 15](#_Toc693069014)

[Coding Standard 7 17](#_Toc1421357264)

[Coding Standard 8 19](#_Toc237764880)

[Coding Standard 9 22](#_Toc759904696)

[Coding Standard 10 25](#_Toc1943653129)

[Defense-in-Depth Illustration 27](#_Toc577212939)

[Project One 28](#_Toc1427947093)

[Revise the C/C++ Standards 28](#_Toc344575553)

[Risk Assessment 28](#_Toc1492655313)

[Automated Detection 28](#_Toc1819842708)

[Automation 28](#_Toc1773944149)

[Summary of Risk Assessments 30](#_Toc1120847820)

[Create Policies for Encryption and Triple A 30](#_Toc1292676221)

[Map the Principles 32](#_Toc327544481)

[Audit Controls and Management 36](#_Toc910245679)

[Enforcement 36](#_Toc872142376)

[Exceptions Process 36](#_Toc928089357)

[Distribution 36](#_Toc2096745443)

[Policy Change Control 37](#_Toc1453807233)

[Policy Version History 37](#_Toc388270021)

[Appendix A Lookups 37](#_Toc1938304688)

[Approved C/C++ Language Acronyms 37](#_Toc1242997430)

## 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. Validate Input Data | Always validate data received from an external source before processing it. This prevents security vulnerabilities like SQL injection or buffer overflow attacks. |
| 2. Heed Compiler Warnings | Compiler warnings alert you to potential issues in your code that could lead to undefined behavior or security vulnerabilities. Always address these warnings. |
| 3. Architect and Design for Security Policies | Ensure that the system's architecture and design align with organizational security policies and best practices. This may include measures like encryption, authentication, and authorization. |
| 4. Keep It Simple | The simpler the code, the easier it is to maintain and secure. Complex code is harder to analyze, more likely to contain errors, and potentially easier to exploit. |
| 5. Default Deny | When setting up permissions and access, the default setting should be to deny access. Only grant permissions explicitly when necessary. |
| 6. Adhere to the Principle of Least Privilege | Users and systems should have the minimum levels of access—or permissions—necessary to perform their functions. This minimizes the risk associated with unauthorized access. |
| 7. Sanitize Data Sent to Other Systems | Always sanitize and validate data before sending it to other systems to prevent data leaks or injection attacks. |
| 8. Practice Defense in Depth | Employ multiple layers of security controls. If one layer is compromised, others are still in place to prevent a breach. |
| 9. Use Effective Quality Assurance Techniques | Employ a combination of techniques like code reviews, unit testing, and automated testing to ensure that the code is secure and works as intended. |
| 10. Adopt a Secure Coding Standard | Adhering to a secure coding standard can help to prevent common security vulnerabilities and helps maintain a consistent codebase, making it easier to manage and secure. |

### 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 | **Name of Standard**: Use Appropriate Data Types  **Rationalize the Standard**: Choosing the correct data types is important for both the performance and correctness of the code. Using inappropriate types can lead to overflow, underflow, or data loss, and may make the program's behavior undefined or unpredictable. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the integer overflow is possible because the maximum value that an **int** can store might be exceeded when squaring the number. |
| #include <iostream>    void printSquare() {  int num = 46341; // sqrt(INT\_MAX) + 1  int square = num \* num; // Overflow  std::cout << "Square: " << square << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant example, a 64-bit integer data type (**int64\_t**) is used to ensure that the value does not overflow. This prevents undefined or unpredictable behavior and makes the code more reliable. |
| #include <iostream>  #include <cstdint>    void printSquare() {  int64\_t num = 46341; // sqrt(INT\_MAX) + 1  int64\_t square = num \* num; // No Overflow  std::cout << "Square: " << square << std::endl;  } |

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

| **Principles(s):**   1. **Validate Input Data:** This principle is directly relevant because using appropriate data types also involves validating the type and size of input data. When you validate data types, you ensure that the received data will fit within the allocated memory and won't cause overflow or underflow. 2. **Heed Compiler Warnings:** Compiler warnings can often alert developers to potential issues related to inappropriate data types, such as possible overflows or type mismatches. 3. **Keep It Simple:** By using the appropriate data types, you are simplifying the code. Making sure that data types match their intended use can reduce the chances of unforeseen complications or vulnerabilities. 4. **Use Effective Quality Assurance Techniques:** Quality assurance techniques, such as code reviews or unit testing, can help identify and fix issues related to data type mismatches or potential overflows/underflows. 5. **Adopt a Secure Coding Standard:** Using appropriate data types is a fundamental aspect of many secure coding standards. Adhering to this best practice ensures consistency, predictability, and security in the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | S923 - Appropriate Data Type Usage | SonarQube's code analysis tool, specifically "S923," can detect data type misuse in C++ code. Integrating this automation into Continuous Integration ensures early issue detection and resolution, improving application quality and reducing remediation costs. This approach can also enhance security and quality by identifying principles, assessing risks, and recommending automation for encryption and Triple-A policies. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Name of Standard**: Use Valid References, Pointers, and Iterators  **Rationalize the Standard**: References, pointers, and iterators provide a way to access elements within a container. Using invalid or uninitialized ones can lead to undefined behavior, including crashes and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the iterator **it** is uninitialized and then dereferenced, leading to undefined behavior. |
| std::vector<int> vec = {1, 2, 3};  std::vector<int>::iterator it;  std::cout << \*it; // Undefined behavior |

| **Compliant Code** |
| --- |
| In this compliant code, the iterator **it** is initialized with the beginning of the vector **vec**. Thus, dereferencing it is valid and results in well-defined behavior. |
| std::vector<int> vec = {1, 2, 3};  std::vector<int>::iterator it = vec.begin();  std::cout << \*it; // Outputs "1" |

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

| **Principles(s):**   1. **Validate Input Data:** Before dereferencing references, pointers, or iterators, it's crucial to validate that they are initialized and point to valid memory locations or elements within a container. This avoids potential vulnerabilities and unpredictable behavior. 2. **Heed Compiler Warnings:** Modern compilers will often issue warnings when they detect uninitialized pointers, references, or iterators being used. It's vital to pay attention to these warnings as they can help in identifying and fixing these issues before they lead to security vulnerabilities. 3. **Keep It Simple:** When using references, pointers, or iterators, it's advisable to keep the operations simple and straightforward. Avoiding complex manipulations can reduce the chances of introducing undefined behavior. 4. **Use Effective Quality Assurance Techniques:** Applying quality assurance practices, such as code reviews, unit testing, and static code analysis, can help in detecting and preventing potential issues related to the misuse of references, pointers, and iterators. 5. **Adopt a Secure Coding Standard:** Ensuring the valid use of references, pointers, and iterators should be part of any secure coding standard. Consistently following this practice will make the codebase more robust against potential vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 13.0 | uninitialized.Undefined | Clang Static Analyzer checks for bugs in C, C++, and Objective-C programs, including uninitialized variables that can lead to undefined behavior. Using it in the DevSecOps pipeline saves time and effort by catching vulnerabilities early in development. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | **Name of Standard**: Use Safe String Handling Functions  **Rationalize the Standard**: Insecure string handling can lead to vulnerabilities like buffer overflows, making it critical to use safe string functions that perform bounds-checking. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the **strcpy** function, which does not perform any bounds-checking, to copy a string into a buffer that is too small. This leads to buffer overflow and undefined behavior. |
| char buffer[10];  strcpy(buffer, "This string is too long!"); |

| **Compliant Code** |
| --- |
| This compliant code uses the C++ standard library's **std::string** class, which handles memory allocation and bounds-checking automatically, making it a safer alternative for string manipulation. |
| std::string str = "This is a safe string"; |

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

| **Principles(s):**   1. **Validate Input Data:** Safe string handling functions inherently validate the length and content of strings before processing. By ensuring that strings fit within their allocated memory, potential vulnerabilities like buffer overflows can be prevented. 2. **Keep It Simple:** Using standard libraries that automatically handle memory allocation and bounds-checking reduces the complexity of string manipulations, making the code simpler and safer. 3. **Sanitize Data Sent to Other Systems:** Even if this specific standard emphasizes reading and managing strings securely, it's an important reminder that when sending these strings to other systems, they should be sanitized and validated. Using secure string functions is a step in this direction. 4. **Practice Defense in Depth:** Using safe string functions is one layer of protection. It's beneficial to employ other security measures, such as input validation, ASLR (Address Space Layout Randomization), and canaries in stack frames, to further defend against potential exploits. 5. **Use Effective Quality Assurance Techniques:** Employing code reviews, static analysis tools, and dynamic testing can ensure that developers are using safe string handling functions throughout the codebase. 6. **Adopt a Secure Coding Standard:** Adopting a secure coding standard will emphasize the use of safe string handling functions across the development team. Such a standard ensures that all developers handle strings securely and consistently. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Common | High | High | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.01 | BUFFER\_OVERRUN | The Clang Static Analyzer is a tool for analyzing source code in C, C++, and Objective-C programs. One of its checkers, "uninitialized.Undefined," can identify uninitialized variables (including references, pointers, and iterators) that are utilized to lead to undefined behavior.    By incorporating the Clang Static Analyzer into the DevSecOps pipeline, particularly during the Continuous Integration (CI) phase, it is possible to catch these coding vulnerabilities early in the development process. This early identification and correction can help save significant amounts of time and effort when debugging later in the development cycle. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **Name of Standard**: Prevent SQL Injection  **Rationalize the Standard**: SQL Injection is a common attack vector that can allow an attacker to interfere with the queries your application makes to its database. It's essential to validate and sanitize user input to prevent this vulnerability. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the SQL query is constructed directly from user input, opening the possibility for SQL Injection if **getUserInput()** returns a malicious string. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  executeSQL(query); |

| **Compliant Code** |
| --- |
| This compliant code uses a prepared statement and binds the username as a parameter, ensuring that it's treated as data rather than executable code. This approach effectively mitigates the risk of SQL Injection. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = ?";  prepareStatement(query);  bindParameter(1, username);  executePreparedStatement(); |

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

| **Principles(s):**   1. **Validate Input Data:** This principle is at the heart of preventing SQL injection. By validating user input and ensuring it adheres to expected patterns or lengths, you can mitigate the risk of malicious input being injected into the SQL query. 2. **Keep It Simple:** Directly embedding user input into SQL queries can lead to complexity and unpredictability. Using prepared statements with bound parameters simplifies the interaction with the database, reducing potential errors and vulnerabilities. 3. **Default Deny:** This mindset involves treating all input as potentially harmful until it's validated. In the context of SQL queries, don't assume user input is safe. Instead, use mechanisms like prepared statements to ensure that the input is treated purely as data. 4. **Adhere to the Principle of Least Privilege:** When setting up database access, ensure that the executing context has only the permissions necessary to perform its function. This limits the potential damage an attacker can do even if they manage to inject malicious SQL. 5. **Sanitize Data Sent to Other Systems:** The database is another system your application interacts with. Ensuring that only sanitized and validated data gets sent to the database helps prevent SQL injection attacks. 6. **Practice Defense in Depth:** Using prepared statements and parameterized queries is one defense against SQL injection. Additionally, other measures like input validation, using low-privilege database accounts, and monitoring for suspicious database activity further increase security. 7. **Use Effective Quality Assurance Techniques:** Regular code reviews, especially focusing on parts of the code that interact with databases, can help spot potential vulnerabilities. Automated security scanning tools and penetration testing can also detect possible SQL injection points. 8. **Adopt a Secure Coding Standard:** A secure coding standard should emphasize the importance of preventing SQL injection by always treating user input with suspicion and using safer methods to interact with databases. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Common | High | High | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQLMap | 2.1 | SQL Injection Vulnerability Scan | SQLMap is an open-source tool for identifying and exploiting SQL Injection flaws. It can also be used for security testing to find weak spots. By running it in a controlled environment, you can confirm the effectiveness of your mitigation strategies. Integrating SQL Injection scans with tools like SQLMap during DevSecOps testing can help ensure code changes don't introduce vulnerabilities and existing mitigations remain effective. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **Name of Standard**: Avoid Memory Leaks and Use Smart Pointers  **Rationalize the Standard**: Memory leaks can lead to the exhaustion of system resources over time, rendering a system unusable. Smart pointers in C++ help manage the memory automatically, reducing the chance of memory leaks. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, dynamic memory is allocated using **new**, but there's no corresponding **delete[]** to deallocate the memory, leading to a memory leak. |
| int\* arr = new int[10];  // ... operations on arr  // Missing delete[] |

| **Compliant Code** |
| --- |
| In this compliant example, a **std::unique\_ptr** smart pointer is used to manage the dynamically allocated array. When **arr** goes out of scope, the memory is automatically deallocated, preventing memory leaks. |
| std::unique\_ptr<int[]> arr(new int[10]);  // ... operations on arr |

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

| **Principles(s):**   1. **Validate Input Data:** This principle relates to ensuring that the size and type of memory allocations are appropriate and won't lead to issues such as buffer overflows. Though not directly about memory leaks, it's crucial to ensure that dynamically allocated memory is well-defined and that operations on it are bounded. 2. **Heed Compiler Warnings:** Modern compilers often provide warnings about potential memory leaks, unmanaged dynamic allocations, or resources that aren't freed. Listening to these warnings can aid in catching and preventing memory-related vulnerabilities. 3. **Keep It Simple:** Using smart pointers like **std::unique\_ptr** simplifies memory management, reducing the chance of errors. The code is clearer, and there's no need to manually manage memory deallocations. 4. **Default Deny:** This mindset is about being proactive and cautious. By defaulting to smart pointers, developers deny potential mishandling of raw pointers, reducing the risk of leaks. 5. **Adhere to the Principle of Least Privilege:** Ensuring that memory is only allocated when necessary and deallocated promptly helps in following this principle. Using smart pointers guarantees that memory isn't accessible beyond its intended scope. 6. **Practice Defense in Depth:** While using smart pointers is a primary defense against memory leaks, other techniques, like bounds-checking and container safety, further ensure memory safety. 7. **Use Effective Quality Assurance Techniques:** Employing techniques like code reviews, static analysis tools, and dynamic analysis tools can help identify and mitigate potential memory leaks and mismanagement. 8. **Adopt a Secure Coding Standard:** Incorporating the use of smart pointers as part of the coding standards ensures that all developers adhere to best practices for memory management in the C++ codebase. This standardization leads to more predictable and secure applications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind is a valuable programming tool that supports memory debugging, memory leak detection, and profiling. One of its most useful features is Memcheck, which can identify where memory is not being correctly deallocated in the code, making it beneficial in detecting memory leaks.    By integrating Valgrind's Memcheck into the DevSecOps pipeline, especially during the testing phase, developers can detect and fix memory leaks in the application at an early stage. This early detection and resolution can prevent performance issues and system resource depletion in production environments. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Name of Standard**: Use Assertions Judiciously  **Rationalize the Standard**: Assertions are commonly used to catch programming errors during development and debugging. They can be used to validate assumptions but should not be used to handle runtime errors or to replace proper error handling in the application. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an assertion is used to check for division by zero. However, assertions are typically disabled in release builds, making this an inappropriate method for handling a runtime error that could still occur. |
| #include <cassert>    void divide(int a, int b) {  assert(b != 0);  std::cout << a / b << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant example, proper runtime error handling is used instead of an assertion to check for division by zero. This ensures that the error is appropriately handled regardless of whether the code is a debug or release build. |
| void divide(int a, int b) {  if (b == 0) {  std::cerr << "Error: Division by zero" << std::endl;  return;  }  std::cout << a / b << std::endl;  } |

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

| **Principles(s):**   1. **Validate Input Data:** Properly validating input data ensures that your assumptions about the data hold true. While assertions can be used for this during the development phase, they should not replace thorough input validation checks that remain active in production code. 2. **Keep It Simple:** Assertions are straightforward mechanisms to validate assumptions during development. However, depending on them for runtime error handling can complicate things, as their behavior varies between debug and release builds. 3. **Practice Defense in Depth:** Assertions can be one of the layers of error checking during the development and debugging phase. However, multiple layers of checks, including thorough error handling mechanisms, should be in place for production code. 4. **Use Effective Quality Assurance Techniques:** Employing techniques like unit testing, code reviews, and automated static analysis can ensure that assertions are used correctly and that proper error-handling mechanisms replace them where necessary in the final product. 5. **Adopt a Secure Coding Standard:** A secure coding standard would emphasize the appropriate use of assertions, clarifying that they are primarily for development and debugging. Such guidelines would ensure developers understand the limitations of assertions and the importance of robust error-handling mechanisms. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | bugprone-assert-side-effect | Clang-Tidy is a valuable tool for C++ programmers that can detect common programming errors. One of its features - the "bugprone-assert-side-effect" check - is particularly noteworthy, as it warns of potential issues when an assertion has a side effect. This can result in different behavior between debug and release builds, which could be problematic. By incorporating Clang-Tidy into the DevSecOps pipeline, developers can be notified of such issues during the static analysis phase, allowing for early detection and remediation. This helps to ensure the code remains secure and robust when released. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | **Name of Standard**: Use Exceptions for Error Handling  **Rationalize the Standard**: Using exceptions for error handling improves the robustness of the code and makes it easier to maintain. Not using exceptions or using them incorrectly can lead to resource leaks, undefined behavior, and makes the code harder to understand. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the function returns a boolean to indicate success or failure, which can make the code harder to read and maintain. It also puts the burden of error checking on the caller. |
| #include <iostream>    bool divide(int a, int b, int& result) {  if (b == 0) {  return false;  }  result = a / b;  return true;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the function throws an exception of type **std::invalid\_argument** when division by zero is attempted. This makes it clear that an error has occurred and allows for centralized error handling.  This approach is in line with modern C++ best practices, ensuring both readability and maintainability while providing a robust mechanism for error handling. |
| #include <iostream>  #include <stdexcept>    int divide(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Division by zero");  }  return a / b;  } |

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

| **Principles(s):**   1. **Validate Input Data:** Validating input and internal state are crucial. Throwing exceptions allows for the centralized handling of erroneous states or inputs, ensuring the application behaves predictably even in the face of unexpected conditions. 2. **Keep It Simple:** Using exceptions simplifies error handling by eliminating the need for extensive error code checking throughout the code. This results in cleaner, more readable, and maintainable code. 3. **Practice Defense in Depth:** Exceptions serve as one layer of error handling. In addition to using exceptions, it's beneficial to employ other preventive measures, like input validation and resource management practices, to ensure code robustness. 4. **Use Effective Quality Assurance Techniques:** Using unit tests and integration tests that intentionally trigger exceptions can ensure that they're properly caught and handled. Regular code reviews can also help in spotting areas where exceptions are either misused or overlooked. 5. **Adopt a Secure Coding Standard:** Adopting a standard that emphasizes the correct use of exceptions for error handling ensures that developers uniformly handle errors. This consistency can reduce the chances of unhandled exceptions leading to vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 14.0 | alpha.core.ExceptionEscape | The Clang Static Analyzer detects C, C++, and Objective-C source code bugs. Its "alpha.core.ExceptionEscape" check prevents exceptions from escaping where they shouldn't. Integrating it into DevSecOps pipelines alerts developers of incorrect exception usage and ensures structured error handling is fully utilized. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-008-CPP | **Name of Standard**: Validate All User Input  **Rationalize the Standard**: Validating user input is crucial for maintaining the security and reliability of an application. Failing to validate input can expose the application to various types of vulnerabilities, such as SQL Injection, Buffer Overflow, and even logic errors that could compromise the application's behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the code directly converts a user-provided string to an integer without validating it first. This can lead to unexpected behavior or errors if the user provides an invalid input. |
| #include <iostream>  #include <string>    void displayUserAge(int age) {  std::cout << "Your age is: " << age << std::endl;  }    int main() {  std::string input;  std::cin >> input;  int age = std::stoi(input);  displayUserAge(age);  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the code first checks whether the conversion from a string to an integer was successful. It then validates that the age is within a reasonable range before proceeding. |
| #include <iostream>  #include <string>  #include <sstream>    void displayUserAge(int age) {  if (age < 0 || age > 150) {  std::cerr << "Invalid age provided." << std::endl;  return;  }  std::cout << "Your age is: " << age << std::endl;  }    int main() {  std::string input;  std::cin >> input;    std::stringstream ss(input);  int age;  if (ss >> age) {  displayUserAge(age);  } else {  std::cerr << "Invalid input. Please enter a number." << std::endl;  }  return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data:** This principle is the main theme of this standard. Ensuring all user inputs undergo rigorous validation before processing them is fundamental to the security of an application. Validation can prevent a myriad of vulnerabilities that arise from processing unexpected or malicious data. 2. **Keep It Simple:** Structuring the code to handle input in a clear and straightforward manner enhances its maintainability and reduces the likelihood of errors. Ensuring validation logic is not overly complex makes it less error prone. 3. **Practice Defense in Depth:** While validating input is a primary defense, other layers of security, such as parameterized database queries to prevent SQL injection or buffer boundary checks, further ensure that even if some malicious data gets past one defense, others are in place to catch it. 4. **Use Effective Quality Assurance Techniques:** Input validation can be verified using a range of QA techniques. Unit tests can be designed to test boundary conditions, unexpected values, or malicious input. Penetration testing can also help identify any weak points in input validation. 5. **Adopt a Secure Coding Standard:** Having a secure coding standard that underlines the importance of input validation ensures that all developers handle user data with the necessary caution and rigor. Consistency in input validation across the application reduces the risk of vulnerabilities. 6. **Architect and Design for Security Policies:** Input validation should be a key consideration in the architecture and design phase of an application, ensuring that security policies around data integrity and validation are effectively implemented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 14.0 | alpha.security.taint.TaintPropagation | The Clang Static Analyzer checks for taint analysis. It can help detect where user input comes from and trace its path through the program to potential sinks like database queries or output operations. Taint analysis can identify areas where user input is used without adequate validation or sanitation, ensuring consistent enforcement throughout the codebase. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | STD-009-CPP | **Name of Standard**: Ensure Thread Safety in Concurrent Programs  **Rationalize the Standard**: Thread safety is crucial for the correct and efficient operation of multi-threaded programs. Failure to ensure thread safety can lead to a range of issues including data corruption, unpredictable behavior, and deadlocks. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, multiple threads are updating a shared resource without any synchronization, which can lead to race conditions and incorrect values. |
| #include <thread>  #include <iostream>    int sharedResource = 0;    void incrementResource() {  for (int i = 0; i < 1000; ++i) {  ++sharedResource;  }  }    int main() {  std::thread t1(incrementResource);  std::thread t2(incrementResource);  t1.join();  t2.join();  std::cout << "Shared Resource Value: " << sharedResource << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, a mutex is used to synchronize access to the shared resource, thereby preventing race conditions and ensuring the correct value for the shared resource. |
| #include <thread>  #include <iostream>  #include <mutex>    int sharedResource = 0;  std::mutex resourceMutex;    void incrementResource() {  for (int i = 0; i < 1000; ++i) {  std::lock\_guard<std::mutex> lock(resourceMutex);  ++sharedResource;  }  }    int main() {  std::thread t1(incrementResource);  std::thread t2(incrementResource);  t1.join();  t2.join();  std::cout << "Shared Resource Value: " << sharedResource << std::endl;  return 0;  } |

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

| **Principles(s):**   1. **Validate Input Data:** While not immediately evident in the context of the example, validating data before processing it in concurrent threads is essential. Concurrent programs with shared resources can become complex and ensuring that input data is valid can prevent further complications. 2. **Keep It Simple:** Thread safety often involves introducing synchronization mechanisms, like mutexes. While these mechanisms are necessary, the overall logic and flow of the program should remain as simple and clear as possible. Complexity can lead to oversight, which in multi-threaded contexts, can be disastrous. 3. **Default Deny:** Assuming that all concurrent operations on shared resources are potentially unsafe can be beneficial. By introducing synchronization mechanisms by default, you ensure that data integrity and safety are maintained. 4. **Practice Defense in Depth:** Beyond just using synchronization primitives, ensure that other protective measures, such as data validation, appropriate thread pooling, and deadlock prevention mechanisms, are in place to cater to different scenarios. 5. **Use Effective Quality Assurance Techniques:** Multi-threaded applications should be rigorously tested for concurrency issues. Techniques such as stress testing, race condition detection tools, and in-depth code reviews can be instrumental in identifying and rectifying threading issues. 6. **Adopt a Secure Coding Standard:** A coding standard that emphasizes thread safety ensures that all developers are aware of and are using best practices when dealing with concurrent programming. This consistent approach can help avoid common pitfalls and vulnerabilities associated with multi-threaded applications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helgrind (part of Valgrind) | 3.17.0 | N/A (Helgrind provides general race condition checks) | Helgrind, a tool within Valgrind, identifies synchronization errors in C, C++, and Fortran programs using POSIX pthreads. It locates race conditions, inappropriate use of POSIX pthreads, and possible deadlocks. With Helgrind, thread safety problems can be highlighted during testing, making it easier for developers to address them and automate the detection of concurrency problems in the DevOps pipeline. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-010-CPP | **Name of Standard**: Properly Manage Resources  **Rationalize the Standard**: Effective resource management is critical for ensuring that an application performs optimally and securely. Failing to properly manage resources like file handles, network connections, or dynamically allocated memory can lead to performance issues, security vulnerabilities, and system instability. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the code writes to a file but doesn't explicitly close the file handle. While the destructor will eventually close the file, explicitly releasing resources is a good practice to prevent potential issues, such as file corruption. |
| #include <fstream>    void writeToFile(const std::string& message) {  std::ofstream file("log.txt", std::ios::app);  file << message;  // Missing file.close()  } |

| **Compliant Code** |
| --- |
| In this compliant example, the code explicitly checks whether the file is open before attempting to write to it. It also explicitly closes the file handle by calling **file.close()**, ensuring that the resource is released as soon as it is no longer needed. |
| #include <fstream>    void writeToFile(const std::string& message) {  std::ofstream file("log.txt", std::ios::app);  if (file.is\_open()) {  file << message;  file.close();  } else {  std::cerr << "Failed to open file" << std::endl;  }  } |

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

| **Principles(s):**   1. **Validate Input Data:** Ensuring that resources like files, network connections, or memory allocations are valid and available before use can prevent many types of vulnerabilities or failures. For instance, checking if a file opens successfully before writing to it is a form of validation. 2. **Keep It Simple:** Structured and straightforward resource management logic enhances readability and maintainability. A clear pattern for acquiring, using, and releasing resources ensures that mistakes are less likely to occur. 3. **Default Deny:** Always start with the assumption that acquiring a resource might fail. This mindset ensures that the code is prepared to handle situations where resources are unavailable or when there are too many resource requests. 4. **Practice Defense in Depth:** Proper resource management is just one layer of application security and performance. Coupled with input validation, error handling, and other security practices, it ensures that the application remains robust against various challenges. 5. **Use Effective Quality Assurance Techniques:** Employ unit tests, stress tests, and resource monitoring to validate that resources are being managed correctly. Static and dynamic analysis tools can help detect resource leaks or other related issues. 6. **Adopt a Secure Coding Standard:** A secure coding standard that emphasizes resource management ensures that developers consistently and properly handle resources, preventing issues related to resource leaks or exhaustion. 7. **Heed Compiler Warnings:** Many modern compilers can produce warnings related to resource handling, such as potential memory leaks or unclosed file handles. Paying attention to and addressing these warnings can prevent resource management issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Memcheck | 3.17.0 | Memory Leak Detection | Memcheck is a tool in Valgrind that detects memory errors like leaks. This helps ensure proper resource management and can be added to testing or DevOps pipelines to catch issues early. It improves software performance and security. |

### 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.

The DevSecOps methodology integrates security practices into the traditional DevOps process, emphasizing the importance of security throughout the software development lifecycle. The diagram illustrates this integration's continuous nature, with each stage building on the previous one. Automation is used at each stage to enforce compliance with defined standards.

* 1. Assess and Plan: Automated tools are employed to assess the current threat landscape, regulatory changes, and potential change impacts. Routine scans are scheduled to identify vulnerabilities, and automated reports prioritize backlog items based on security risks.
  2. Design: Security as Code (SAC) tools are used to ensure adherence to security standards during the design phase. Automated security design review tools confirm that designs align with security requirements and best practices.
  3. Build: Automated security checks are integrated into the build process to validate trusted repositories, authenticate open-source components, and ensure that built code is free from vulnerabilities. Static Application Security Testing (SAST) tools are used for code vulnerability scans.
  4. Verify and Test (Pre-production): Automated testing tools confirm the absence of vulnerabilities or compliance breaches before production deployment. Dynamic Application Security Testing (DAST) tools scrutinize running applications for vulnerabilities in pre-production.
  5. Transition and Health Check (Production): Automated health checks and penetration tests are executed before entering the production environment to ensure system robustness. Deployment processes are refined with integrated security checks to deploy secured and vetted code.
  6. Monitor and Detect: Using tools like SIEM, constant system monitoring detects security breaches or vulnerabilities. Automated intrusion detection systems establish instantaneous alerts for unusual activities.
  7. Maintain and Stabilize: After any security incident, the system auto-assesses departures from the security baseline and initiates stabilization efforts. Backup and recovery protocols are streamlined and automated to revert the system to a secure state if compromised.
  8. Respond: Automated mechanisms counteract attacks, deactivate impacted services, and initiate rollback actions for system stability when security incidents arise. Automated incident response strategies ensure rapid threat containment and mitigation.

By incorporating these enhancements into the existing DevOps process, the security aspect of software development is strengthened at every stage. A DevSecOps approach, combined with automation, ensures that security is woven into every facet of software development and operations, resulting in more robust and compliant systems.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | High | High | L1 |
| STD-003-CPP | Critical | Common | High | High | L1 |
| STD-004-CPP | Critical | Common | High | High | L1 |
| STD-005-CPP | High | Likely | Medium | Medium | L2 |
| STD-006-CPP | High | Possible | Medium | Medium | L2 |
| STD-007-CPP | Medium | Probable | Medium | Low | L3 |
| STD-008-CPP | High | Likely | Medium | High | L1 |
| STD-009-CPP | High | Likely | High | Medium | L1 |
| STD-010-CPP | Medium | Moderate | Medium | Low | 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 in rest | To safeguard sensitive data stored in a database or on a disk, encryption at rest is used to encrypt the data when it is not in use. Our policy mandates that all sensitive data stored in our systems must be encrypted at rest with industry-standard techniques. This ensures that even if the storage medium or data backup is compromised, the data remains inaccessible without the decryption key. Before storing any sensitive data, it should be encrypted. It is also essential to encrypt any backup copies or replicas. Protecting data at rest is crucial as it ensures that the data remains secure even if physical security controls fail or a data breach occurs. |
| Encryption at flight | Encryption in flight, also known as encryption in transit, protects data while it is being transferred between systems, such as over the internet or through a private network. Our policy requires all sensitive data transferred between systems, whether internally or externally, to be encrypted in flight. To achieve this, we recommend implementing TLS (Transport Layer Security) or other secure protocols for any data transfer. End-to-end encryption should also be ensured for maximum security. This ensures that data remains confidential and unchanged during transmission. |
| Encryption in use | Encryption refers to protecting data that's actively being used or processed. All sensitive data must remain encrypted even when processed or accessed by applications. Techniques like homomorphic encryption, which allows computations on encrypted data, should be employed. Protecting data during processing ensures it remains secure throughout its lifecycle, even when accessed or manipulated. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user, system, or application. Users, systems, or applications must prove their identity to gain access to resources. Implementing strong authentication mechanisms such as multi-factor authentication (MFA) and single sign-on (SSO) is essential. It is also vital to regularly review and update authentication protocols. The reason for this is to prevent unauthorized access by ensuring that a user's or system's identity is confirmed before granting access to resources. |
| Authorization | Authorization is the process of determining which actions, resources, or services an authenticated user can access. To ensure security, it is recommended that users, systems, or applications are given the minimum level of privilege required to perform their tasks. One way to achieve this is to use role-based access controls (RBAC) to manage user permissions. It is essential to review and update permissions as necessary regularly. This approach limits the potential damage of a security breach by ensuring that users only have access to the required resources. |
| Accounting | It involves tracking and recording user activities to monitor and review their actions. Per the policy, all user actions and system activities must be logged and regularly reviewed. To implement it, centralized logging solutions should be set up, logs are stored securely, and regular audits should be conducted. Logging and monitoring user actions is to detect suspicious activities, ensure accountability, and assist in investigations after security incidents. |

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

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

### Map the Principles

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

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

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

**Coding Standard: Use Appropriate Data Types**

**Mapped Principles:** 1, 2, 4, 9, 10

**Justification:**

**1. Validate Input Data:** By using appropriate data types, you inherently validate data to fit within the constraints of that data type, preventing overflows and underflows.

**2. Heed Compiler Warnings:** Data type mismatches or unsafe conversions often trigger compiler warnings.

**4. Keep It Simple:** Choosing the correct data type makes the code straightforward and reduces complexity.

**9. Use Effective Quality Assurance Techniques:** Proper data type usage can be enforced and checked through QA techniques like code reviews.

**10. Adopt a Secure Coding Standard:** Following a secure coding standard includes using data types correctly to prevent vulnerabilities.

**Coding Standard: Use Valid References, Pointers, and Iterators**

**Mapped Principles:** 1, 2, 4, 9, 10

**Justification:**

**1. Validate Input Data:** Ensuring pointers and references are valid before dereferencing is a form of input validation.

**2. Heed Compiler Warnings:** Dereferencing invalid pointers or using invalid iterators can generate compiler warnings.

**4. Keep It Simple:** Using valid references and pointers prevents complications arising from unexpected behavior.

**9. Use Effective Quality Assurance Techniques:** Proper usage can be checked during QA activities like static analysis.

**10. Adopt a Secure Coding Standard:** A secure coding standard would emphasize the importance of using valid pointers, references, and iterators.

**Coding Standard: Use Safe String Handling Functions**

**Mapped Principles:** 1, 4, 7, 8, 9, 10

**Justification:**

**1. Validate Input Data:** Safe string functions inherently validate string lengths and boundaries.

**4. Keep It Simple:** Utilizing standard safe string functions reduces code complexity.

**7. Sanitize Data Sent to Other Systems:** Safe string functions ensure data is sanitized.

**8. Practice Defense in Depth:** Safe string functions provide an essential layer of defense against buffer overflows.

**9. Use Effective Quality Assurance Techniques:** QA can validate the usage of safe functions over unsafe alternatives.

**10. Adopt a Secure Coding Standard:** Proper string handling should be part of any secure coding standard.

**Coding Standard: Prevent SQL Injection**

**Mapped Principles:** 1, 4, 5, 6, 7, 8, 9, 10

**Justification:**

**1. Validate Input Data:** SQL injection prevention starts with validating user input.

**4. Keep It Simple:** Parameterized queries and prepared statements simplify and standardize data-to-SQL interaction.

**5. Default Deny:** Assume any input can be harmful and treat it accordingly.

**6. Adhere to the Principle of Least Privilege:** Limit database permissions.

**7. Sanitize Data Sent to Other Systems:** Ensure any data sent to the database is sanitized.

**8. Practice Defense in Depth:** Multiple defenses, such as input validation and parameterized queries, ensure security.

**9. Use Effective Quality Assurance Techniques:** Regular reviews and testing ensure SQL queries are safe.

**10. Adopt a Secure Coding Standard:** A standard would emphasize proper methods to prevent SQL injection.

**Coding Standard: Avoid Memory Leaks and Use Smart Pointers**

**Mapped Principles:** 1, 2, 4, 5, 6, 8, 9, 10

**Justification:**

**1. Validate Input Data:** Ensure memory allocations and releases are balanced.

**2. Heed Compiler Warnings:** Memory leak warnings can arise during compilation.

**4. Keep It Simple:** Smart pointers simplify memory management.

**5. Default Deny:** Assume that memory management can fail and handle it.

**6. Adhere to the Principle of Least Privilege:** Limit the scope of dynamic memory.

**8. Practice Defense in Depth:** Smart pointers are one layer; other practices like validation further ensure safety.

**9. Use Effective Quality Assurance Techniques:** Memory management can be validated during QA.

**10. Adopt a Secure Coding Standard:** Proper memory management should be a cornerstone of secure coding standards.

**Coding Standard: Use Assertions Judiciously**

**Mapped Principles:** 1, 4, 8, 9, 10

**Justification:**

**1. Validate Input Data:** Assertions can validate certain conditions during development.

**4. Keep It Simple:** Assertions, when used correctly, make code assumptions explicit.

**8. Practice Defense in Depth:** Assertions are a primary defense during development; runtime checks provide added security.

**9. Use Effective Quality Assurance Techniques:** The appropriate use of assertions can be verified during QA.

**10. Adopt a Secure Coding Standard:** The standard should guide the proper use of assertions.

**Coding Standard: Use Exceptions for Error Handling**

**Mapped Principles:** 1, 4, 8, 9, 10

**Justification:**

**1. Validate Input Data:** Exceptions handle unexpected or erroneous inputs or conditions.

**4. Keep It Simple:** Using exceptions streamlines error handling.

**8. Practice Defense in Depth:** Exceptions provide a clear layer of error handling; additional layers like input validation add depth.

**9. Use Effective Quality Assurance Techniques:** Testing can ensure exceptions are properly caught and handled.

**10. Adopt a Secure Coding Standard:** Proper error handling, including exceptions, is crucial for a secure coding standard.

**Coding Standard: Validate All User Input**

**Mapped Principles:** 1, 3, 4, 8, 9, 10

**Justification:**

**1. Validate Input Data:** This principle is the core of this standard.

**3. Architect and Design for Security Policies:** Designing systems to expect and validate all input is foundational.

**4. Keep It Simple:** Clear and structured input validation logic enhances maintainability.

**8. Practice Defense in Depth:** Input validation is a primary defense; additional mechanisms provide further security.

**9. Use Effective Quality Assurance Techniques:** Testing and reviews can ensure input validation is thorough.

**10. Adopt a Secure Coding Standard:** Input validation is a foundational part of secure coding.

**Coding Standard: Ensure Thread Safety in Concurrent Programs**

**Mapped Principles:** 1, 4, 5, 8, 9, 10

**Justification:**

**1. Validate Input Data:** Ensure data accessed across threads is consistently valid.

**4. Keep It Simple:** Clear synchronization logic ensures maintainability.

**5. Default Deny:** Assume concurrent operations are potentially unsafe.

**8. Practice Defense in Depth:** Thread safety mechanisms like mutexes are one defense; other checks like input validation add depth.

**9. Use Effective Quality Assurance Techniques:** Concurrency issues can be detected during QA.

**10. Adopt a Secure Coding Standard:** Thread safety should be emphasized in secure coding guidelines.

**Coding Standard: Properly Manage Resources**

**Mapped Principles:** 1, 2, 4, 5, 8, 9, 10

**Justification:**

**1. Validate Input Data:** Ensure resources are available and valid before use.

**2. Heed Compiler Warnings:** Resource-related warnings can arise during compilation.

**4. Keep It Simple:** Explicit resource management clarifies code logic.

**5. Default Deny:** Assume that resource acquisition might fail.

**8. Practice Defense in Depth:** Proper resource management is a primary defense; other mechanisms add depth.

**9. Use Effective Quality Assurance Techniques:** Resource management can be validated during QA.

**10. Adopt a Secure Coding Standard:** Effective resource management is foundational for secure coding standards.

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 |  |
| 1.1 | 10/07/2023 | Add 10 Coding Standards | Rui Costa |  |
| [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 |