**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Ensuring all input data is validated to prevent malicious data from compromising the system. This involves checking for data type, length, format, and range. |
| 1. Heed Compiler Warnings | Treating compiler warnings seriously and addressing them promptly can prevent potential vulnerabilities and ensure code quality. |
| 1. Architect and Design for Security Policies | Designing systems with security in mind from the outset, incorporating security policies into the architecture and design phases. |
| 1. Keep It Simple | Complexity often leads to security vulnerabilities. Simple, clean code is easier to review and less likely to contain security issues. |
| 1. Default Deny | Only allowing actions that are explicitly permitted and denying all others by default enhances security by minimizing unnecessary access. |
| 1. Adhere to the Principle of Least Privilege | Limiting access rights for users to the bare minimum necessary to perform their work reduces the potential impact of a compromise. |
| 1. Sanitize Data Sent to Other Systems | Ensuring data sent to external systems is sanitized to prevent the propagation of vulnerabilities. |
| 1. Practice Defense in Depth | Employing multiple layers of security measures to protect information and resources in the event one layer fails. |
| 1. Use Effective Quality Assurance Techniques | Implementing thorough testing and QA processes to identify and rectify security issues before deployment. |
| 1. Adopt a Secure Coding Standard | Following a set of guidelines for secure coding to reduce the introduction of vulnerabilities. |

### 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 | Strong Type Checking: Strong type checking is essential for ensuring that variables and functions operate on data of an expected type. This prevents a range of errors and vulnerabilities, including buffer overflows and type mismatches, which can lead to undefined behavior and potential security risks. |

| **Noncompliant Code** |
| --- |
| This block demonstrates an implicit type conversion from **float** to **int**, which can lead to loss of precision and unexpected behavior, posing a risk when precise value handling is required for security decisions. |
| // Noncompliant: Implicit type conversion from float to int  float f = 1.23;  int i = f; // Implicit conversion loses the decimal part, potentially leading to unexpected behavior |

| **Compliant Code** |
| --- |
| This code explicitly converts a **float** to an **int** using **static\_cast**, making the type conversion intentional and clear, thus avoiding the risks associated with implicit conversions. |
| // Compliant: Explicit type conversion with static\_cast  float f = 1.23;  int i = static\_cast<int>(f); // Explicit conversion makes the programmer's intention clear and avoids accidental loss of information |

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

| **Principles(s):** The "Fail Securely" principle is about anticipating and safely handling potential errors or exceptions that may occur during the execution of a program. In the context of STD-001-CPP, the principle applies to type conversions, where implicit conversions can introduce risks, such as loss of precision or unexpected behavior, which might lead to vulnerabilities or logic errors. By using **‘static\_cast** for explicit type conversion, as shown in the compliant code example, the developer makes their intentions clear, reducing the risk of accidental errors. This approach ensures that if a type conversion can't be done safely, the issue is caught at compile time rather than leading to unpredictable behavior at runtime. Thus, adhering to "Fail Securely" by employing strong type checking and explicit conversions contributes to the overall resilience and security of the application. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium - Enforcing strong type checking and explicit conversions can be implemented with minimal changes to the code. | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analysis Tool | 1.0 | TypeSafetyCheck | This tool scans the codebase for instances of implicit type conversions and flags them for review, ensuring that all type conversions are explicit and conform to the strong type checking standard. |

**Coding Standard 2**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validate Data Range - Validating the range of data values is crucial to prevent errors and vulnerabilities that may arise from unexpected or malicious data inputs. Ensuring that data values fall within expected boundaries can prevent issues such as buffer overflows, integer overflows, and logic errors that could compromise system security. |

| **Noncompliant Code** |
| --- |
| This code accepts user input for age without validating its range. An out-of-range value could lead to unexpected behavior or vulnerabilities, especially if used in subsequent calculations or array indexing. |
| // Noncompliant: Accepting user input without validating the range  int getUserAge() {  int age;  std::cout << "Enter your age: ";  std::cin >> age;  return age; // No validation for age value  } |

| **Compliant Code** |
| --- |
| This revised code validates the user's age to ensure it falls within a reasonable and expected range (0-120). It prevents processing of invalid data that could potentially lead to errors or vulnerabilities. |
| // Compliant: Validating user input to ensure it falls within a reasonable range  int getUserAge() {  int age;  do {  std::cout << "Enter your age (0-120): ";  std::cin >> age;  } while (age < 0 || age > 120);  return age;  } |

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

| **Principles(s):** The "Validate Input Data" principle is fundamental in secure coding practices. It emphasizes that all input data, regardless of its source, should be considered untrusted until validated. The principle advocates for checking data for correctness, relevance, and security to ensure it does not contain anything that could lead to vulnerabilities or errors within the system. In the context of STD-002-CPP, this principle is applied by ensuring that the age value provided by the user falls within a predefined, logical range (0-120). This not only prevents potential errors from unexpected values but also mitigates security risks associated with unvalidated input, such as buffer overflows or integer overflows, which could compromise the system's security. By adhering to this principle, developers can significantly reduce the risk of vulnerabilities arising from improperly validated input data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low - Implementing range checks is straightforward and requires minimal changes. | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Dynamic Analysis Tool | 2.0 | InputValidationCheck | This tool dynamically analyzes the application during runtime to detect and report instances where user inputs and data values are not validated for range, helping identify potential vulnerabilities that could be exploited. |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Safe String Handling - Proper string handling is crucial to avoid vulnerabilities such as buffer overflows, format string vulnerabilities, and injection attacks. Ensuring strings are correctly managed, validated, and sanitized protects the application from various exploits that could compromise security. |

| **Noncompliant Code** |
| --- |
| This code uses **strcpy**, which does not check the size of the destination buffer, potentially leading to a buffer overflow if **src** is larger than **dest**. |
| // Compliant: Safe string copy using bounded copy function  void copyString(char \*dest, const char \*src, size\_t size) {  strncpy(dest, src, size); // Safer: limits the number of copied characters  dest[size - 1] = '\0'; // Ensure null-termination  } |

| **Compliant Code** |
| --- |
| This revised code uses **strncpy** to limit the number of characters copied to the size of the destination buffer and explicitly null-terminates the string, preventing buffer overflow. |
| // Compliant: Safe string copy using bounded copy function  void copyString(char \*dest, const char \*src, size\_t size) {  strncpy(dest, src, size); // Safer: limits the number of copied characters  dest[size - 1] = '\0'; // Ensure null-termination  } |

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

| **Principles(s):** The "Keep It Simple" principle is pivotal in secure coding as it suggests that the simpler the code, the easier it is to understand, maintain, and secure. In the context of STD-003-CPP, applying this principle means using string handling functions that inherently include bounds checking, like **‘strncpy’**, instead of more error-prone functions like **‘strcpy’**. This approach reduces the complexity involved in manually ensuring string operations stay within buffer limits, thereby decreasing the likelihood of introducing vulnerabilities such as buffer overflows. By adhering to simplicity, developers can more effectively manage and validate string operations, ensuring they contribute to the overall security and robustness of the application. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Moderate | Medium - Retrofitting safe string handling into an existing codebase may require significant refactoring. | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analysis Tool | 1.2 | SafeStringFunctionsCheck | This tool scans the codebase for unsafe string manipulation functions (like **strcpy**, **sprintf**, etc.) and suggests safer alternatives (like **strncpy**, **snprintf**, etc.), helping developers avoid common pitfalls in string handling. |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Preventing SQL Injection - SQL Injection vulnerabilities allow attackers to interfere with the queries that an application makes to its database. It's a serious vulnerability that can lead to the unauthorized viewing of data, data manipulation, and even administrative operations on the database. Preventing SQL injection involves validating all input data and using secure database access mechanisms like prepared statements. |

| **Noncompliant Code** |
| --- |
| This code directly concatenates user input into an SQL query string, making it susceptible to SQL injection if the user input includes SQL commands. |
| // Noncompliant: Constructing SQL query using string concatenation with user input  std::string constructQuery(std::string user\_input) {  return "SELECT \* FROM users WHERE user\_name = '" + user\_input + "';";  } |

| **Compliant Code** |
| --- |
| This code uses prepared statements with parameterized queries to safely include user input in SQL queries. This method ensures that user input is handled as data, not as part of the SQL command, preventing SQL injection. |
| // Compliant: Using prepared statements to safely create SQL queries  std::string constructQuery(std::string user\_input) {  // Assuming `db` is a database connection object  auto stmt = db.prepareStatement("SELECT \* FROM users WHERE user\_name = ?");  stmt.setString(1, user\_input); // Safely sets the user input in the query  return stmt.executeQuery();  } |

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

| **Principles(s):** The principle of "Validate Input Data" is central to secure coding practices, insisting that all input data, irrespective of its source, be considered untrusted until proven otherwise through rigorous validation. In the context of STD-004-CPP, adhering to this principle means ensuring that user inputs are not only validated for their content but also securely handled to prevent them from being executed as part of SQL commands. The use of prepared statements with parameterized queries exemplifies this principle in action, as it effectively separates data from code, treating user input strictly as data and not executable code, thereby mitigating the risk of SQL injection attacks. This approach ensures that the system remains resilient against one of the most prevalent and dangerous web application vulnerabilities by validating and sanitizing all user-generated inputs before they interact with the database. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low to Medium - Converting to prepared statements may require significant changes in the way database queries are handled, but it's straightforward in terms of complexity. | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Scanning Tool | 2.5 | SQLInjectionCheck | This tool scans the code for patterns that indicate dynamic SQL query construction with user-controlled input and flags them for review. It suggests modifications to use prepared statements or other safe query construction methods, aiding developers in mitigating SQL injection risks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Secure Memory Management - Secure memory management is crucial to prevent vulnerabilities such as buffer overflows, use-after-free errors, and memory leaks, which can lead to undefined behavior, system crashes, or security breaches. Ensuring that memory is allocated, used, and freed correctly protects the integrity of the application and the underlying system. |

| **Noncompliant Code** |
| --- |
| This code allocates a fixed-size buffer for **dest** and then copies **src** into it without checking the length of **src**. This can lead to a buffer overflow if **src** is longer than 9 characters (plus the null terminator). |
| // Noncompliant: Allocation without proper size calculation, leading to potential buffer overflow  char\* copyString(const char\* src) {  char\* dest = new char[10]; // Fixed size allocation  strcpy(dest, src); // Dangerous: src might exceed 10 characters  return dest;  } |

| **Compliant Code** |
| --- |
| This code calculates the required buffer size based on the length of **src** and allocates sufficient memory for **dest**. It then uses **strncpy** to safely copy the string, including a null terminator, preventing buffer overflow. |
| // Compliant: Safe memory allocation and copying with size checks  char\* copyString(const char\* src) {  size\_t len = strlen(src) + 1; // +1 for the null terminator  char\* dest = new char[len]; // Allocate enough memory  strncpy(dest, src, len); // Copy with boundary check  dest[len - 1] = '\0'; // Ensure null-termination  return dest;  } |

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

| **Principles(s):** "Practice Defense in Depth" principle involves implementing a comprehensive approach to security by incorporating various defensive strategies to safeguard the system at different levels. In the realm of secure memory management, this principle is applied by ensuring that there are multiple checks and balances in place to prevent memory-related vulnerabilities such as buffer overflows and use-after-free errors. This includes validating the length of data before memory operations, using safe memory manipulation functions like **‘strncpy’** instead of **‘strcpy’**, and ensuring proper allocation and deallocation of memory. By adopting a defense-in-depth approach to memory management, developers can create more resilient applications that are protected against a wide range of attack vectors, thus maintaining the integrity and security of the application and the underlying system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Moderate | Medium - Retrofitting safe memory management into existing codebases can require significant effort but is essential for security. | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Memory Safety Analysis Tool | 3.1 | BufferOverflowCheck | This advanced analysis tool scans the codebase for patterns that may lead to buffer overflows, use-after-free errors, and memory leaks. It flags risky memory operations and suggests safer alternatives or practices, aiding developers in reinforcing memory safety in their applications. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Proper Use of Assertions - Assertions are used to ensure that certain assumptions about the program's state hold true at runtime, which is crucial for catching bugs and logical errors during the development phase. However, they should not be used to control program flow or handle runtime errors in production code, as assertions may be disabled in release builds, leading to unhandled errors. |

| **Noncompliant Code** |
| --- |
| This code uses an assertion to validate user input, which is not suitable for production environments where assertions might be disabled, leaving no input validation in place. |
| // Noncompliant: Using an assertion to validate user input  void processUserInput(int userInput) {  assert(userInput >= 0 && userInput <= 100); // Inappropriate use of assert for input validation  // Process the input  } |

| **Compliant Code** |
| --- |
| This code checks the user input and throws an exception if the input is out of the expected range. This ensures that input validation is always performed, even if assertions are disabled. |
| // Compliant: Properly handling user input with error handling  void processUserInput(int userInput) {  if (userInput < 0 || userInput > 100) {  throw std::out\_of\_range("Input must be between 0 and 100.");  }  // Process the input  } |

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

| **Principles(s):** The principle of "Fail Securely" is about designing systems to handle errors and exceptions in a secure manner, preventing error conditions from leading to security vulnerabilities. In the context of STD-006-CPP, using assertions for input validation in production code violates this principle because assertions may not work in release builds, potentially leaving the system without proper input validation. This could lead to unhandled errors that might compromise the system's security. Instead, the compliant code demonstrates the "Fail Securely" principle by using proper error handling mechanisms (in this case, throwing an exception) to ensure that the system responds to invalid input in a secure manner, maintaining the integrity and security of the application even in failure scenarios. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | High | Low - Replacing assertions with proper error handling mechanisms is straightforward and improves code reliability. | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Linting Tool | 1.4 | MisusedAssertionsCheck | This tool examines the code for assertions used outside of debugging contexts, particularly for input validation or control flow. It highlights these instances and recommends more appropriate error handling strategies, helping developers maintain code correctness and security. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Secure Exception Handling - Proper exception handling is essential for maintaining application stability and preventing security vulnerabilities. Exceptions should be caught and handled in a way that doesn't expose sensitive information or lead to inconsistent application states. It's also important to catch exceptions at the right level of granularity and to clean up resources appropriately. |

| **Noncompliant Code** |
| --- |
| This code uses a generic catch-all exception handler, which can obscure the source of errors and may not allow for specific handling of different exception types, potentially leading to security risks if some exceptions require special cleanup or logging. |
| // Noncompliant: Catching all exceptions as a generic exception, potentially missing specific handling needs  void processData() {  try {  // Code that might throw exceptions  } catch (...) { // Catching all exceptions  std::cerr << "An error occurred." << std::endl;  }  } |

| **Compliant Code** |
| --- |
| This code catches specific exception types, allowing for tailored handling of different error conditions, including appropriate cleanup, logging, and rethrowing when necessary. This approach ensures that sensitive information is not leaked and that the application can recover gracefully from errors. |
| // Compliant: Catching specific exceptions allows for appropriate handling  void processData() {  try {  // Code that might throw exceptions  } catch (const std::out\_of\_range& e) {  std::cerr << "Out of Range error: " << e.what() << std::endl;  // Handle Out of Range error  } catch (const std::exception& e) {  std::cerr << "Standard error: " << e.what() << std::endl;  // Handle other standard errors  } catch (...) {  std::cerr << "Unknown error occurred." << std::endl;  // Perform necessary cleanup  throw; // Rethrow to allow further handling up the call stack  }  } |

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

| **Principles(s):** The "Principle of Least Privilege" in the context of exception handling encourages catching exceptions at the appropriate level where they can be handled correctly and securely. This prevents the propagation of errors to parts of the system that might not be equipped to deal with them securely, thereby minimizing the potential for security breaches. In STD-007-CPP, using catch-all exception handlers (catch (...)) contradicts this principle as it does not discriminate between exception types and might lead to generic, less secure error handling. Conversely, catching specific exceptions allows for tailored and secure handling at the right level of the application, adhering to the "Principle of Least Privilege" by ensuring that only the necessary information is exposed and only the requisite actions are taken in response to an exception, thereby maintaining application stability and security.Top of Form |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Moderate | Medium - Refactoring code to handle exceptions properly requires a good understanding of the possible exceptions and how to handle them securely. | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Security Tool | 2.2 | ExceptionHandlingCheck | This tool analyzes the codebase for exception handling patterns, identifying generic catch blocks and recommending more specific handling based on the exceptions that can be thrown by the called functions. It helps developers refine their error handling to be more secure and informative. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Validation** | STD-008-CPP | Comprehensive Input Validation - Proper input validation is critical for application security, ensuring that only appropriately formatted data is processed by the application. This practice helps prevent common security issues such as SQL injection, cross-site scripting (XSS), command injection, and more. Validation should occur at the earliest point of entry into the system and should include checks for data type, length, format, and range as applicable. |

| **Noncompliant Code** |
| --- |
| This code snippet directly uses user-provided input (**filePath**) to open a file without any validation, making it susceptible to directory traversal or file inclusion vulnerabilities. |
| // Noncompliant: Accepting unvalidated file paths from user input  void openFile(const std::string& filePath) {  std::ifstream file(filePath); // Risky: filePath might include malicious paths  // Process the file  } |

| **Compliant Code** |
| --- |
| This code includes a validation function (**isValidPath**) that checks the **filePath** against a set of validation rules before the file is opened. This prevents the application from processing malicious file paths. |
| // Compliant: Validating file paths before usage  bool isValidPath(const std::string& filePath) {  // Implement validation logic, e.g., checking for allowed directory paths, file extensions, etc.  return true; // Placeholder return value  }  void openFile(const std::string& filePath) {  if (!isValidPath(filePath)) {  throw std::invalid\_argument("Invalid file path");  }  std::ifstream file(filePath); // Safer: filePath is validated before use  // Process the file  } |

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

| **Principles(s):** The "Validate Input Data" principle is crucial in secure coding practices, advocating for the scrutiny of all input data to ensure it adheres to expected formats, types, and ranges, thus considered safe to process. In the context of STD-008-CPP, this principle is applied through the validation of **‘filePath’** to prevent directory traversal or file inclusion vulnerabilities. By implementing a validation function (**‘isValidPath’**), the code ensures that the **‘filePath’** is scrutinized against predefined rules before being used to open a file. This proactive measure not only guards against potential security threats but also aligns with the best practice of treating all user-supplied data as untrusted until validated. This approach is instrumental in safeguarding the application from a wide array of input-related vulnerabilities, thereby maintaining the integrity and security of the system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium - Implementing thorough input validation can be straightforward for new developments but might require significant refactoring for existing applications. | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Input Validation Testing Tool | 1.0 | PathValidationCheck | This tool automatically tests input validation mechanisms within the application, particularly focusing on file path validations, user inputs, and other entry points. It identifies inputs that bypass validation checks, helping developers strengthen their validation routines. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Cryptographic Practices** | STD-009-CPP | Secure Cryptographic Practices - Employing cryptography securely is fundamental in protecting sensitive data. This includes using strong, up-to-date cryptographic algorithms, proper key management, and understanding the context in which cryptography is applied. Misuse of cryptography can lead to vulnerabilities that compromise the confidentiality, integrity, and availability of data. |

| **Noncompliant Code** |
| --- |
| This code uses MD5 for password hashing, which is insecure and vulnerable to attacks. Modern applications require stronger hashing algorithms designed for password protection, like bcrypt, Argon2, or PBKDF2. |
| // Noncompliant: Using a deprecated cryptographic function for password hashing  std::string hashPassword(const std::string& password) {  // Example using a simple, insecure hash function (e.g., MD5, SHA-1)  return md5(password); // MD5 is not secure for password hashing  } |

| **Compliant Code** |
| --- |
| This code uses bcrypt, a secure hashing function designed for password hashing, which includes salting and multiple rounds of hashing to protect against brute-force and rainbow table attacks. |
| // Compliant: Using a secure hash function with proper salting for passwords  std::string hashPassword(const std::string& password, const std::string& salt) {  // Example using a secure hash function designed for passwords  return bcrypt(password + salt); // bcrypt is a secure choice for password hashing  } |

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

| **Principles(s):** The "Defense in Depth" principle in secure cryptographic practices underscores the need for a comprehensive approach to data security, which includes using robust and current cryptographic algorithms, effective key management strategies, and a deep understanding of the cryptographic context. The noncompliant code example demonstrates a reliance on a single, outdated cryptographic function (MD5) for password hashing, which is insufficient in the face of modern threats. In contrast, the compliant code leverages bcrypt, a stronger hashing algorithm with built-in mechanisms like salting and multiple hashing rounds, which align with "Defense in Depth" by adding multiple hurdles for attackers, such as resistance to brute-force and rainbow table attacks. This multi-faceted approach to cryptography ensures that even if attackers can bypass one security layer, additional protections are in place to safeguard sensitive data, thereby maintaining its confidentiality, integrity, and availability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Moderate | Medium - Implementing or migrating to secure cryptographic practices may require significant changes, especially if legacy systems are involved. | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cryptography Security Analysis Tool | 2.0 | CryptoAlgorithmCheck | This tool scans the codebase for the use of cryptographic functions, identifying the use of weak or deprecated algorithms and recommending secure alternatives. It aids developers in adhering to best practices in cryptography, ensuring the secure handling of sensitive data. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Concurrency** | STD-010-CPP | Secure Concurrency Management - Proper management of concurrent operations is essential to prevent race conditions, deadlocks, and other concurrency-related vulnerabilities that can compromise data integrity and application stability. Secure concurrency management involves using thread-safe functions, proper synchronization mechanisms, and understanding the concurrency model to ensure safe access to shared resources. |

| **Noncompliant Code** |
| --- |
| This code snippet increments a shared counter without any form of synchronization, leading to a race condition where the final value of **sharedCounter** might not reflect the actual number of increments due to concurrent modifications. |
| // Noncompliant: Unsynchronized access to a shared resource in a multi-threaded context  int sharedCounter = 0;  void incrementCounter() {  sharedCounter++; // Risky: simultaneous access by multiple threads can lead to a race condition  } |

| **Compliant Code** |
| --- |
| This code uses a **std::mutex** and a **std::lock\_guard** to ensure that only one thread can increment **sharedCounter** at a time, preventing race conditions. |
| #include <mutex>  std::mutex mtx; // Mutex for synchronizing access to sharedCounter  int sharedCounter = 0;  void incrementCounter() {  std::lock\_guard<std::mutex> lock(mtx); // Locks the mutex, ensuring exclusive access to sharedCounter  sharedCounter++; // Safe: protected by mutex  } |

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

| **Principles(s):** The "Keep It Simple" principle is especially crucial in the context of concurrency, where the interactions between threads or processes can quickly become complicated and hard to predict, leading to issues like race conditions or deadlocks. The noncompliant code example demonstrates a common pitfall in concurrent programming—modifying shared state without synchronization—resulting in a race condition. In contrast, the compliant code illustrates a straightforward and effective approach to ensuring thread safety by using a mutex (**‘std::mutex’**) and a lock guard (**‘std::lock\_guard<std::mutex>’**) to manage access to the shared resource (**‘sharedCounter’**). This method encapsulates the complexity of thread synchronization, making the code easier to understand, maintain, and less prone to concurrency-related vulnerabilities. By adhering to the "Keep It Simple" principle, developers can write more secure, stable, and reliable concurrent applications. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Moderate | Medium - Fixing concurrency issues might require significant architectural changes, especially if the application was not designed with thread safety in mind. | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Concurrency Analysis Tool | 1.3 | ThreadSafetyCheck | This tool analyzes the application's concurrency model, identifying unsynchronized access to shared resources and potential deadlocks. It suggests thread-safe practices and synchronization mechanisms to mitigate risks associated with concurrent operations. |

#### Coding Standard 11

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Secure API Interaction** | STD-011-CPP | "Secure API Interaction" emphasizes the importance of implementing security measures in the way applications communicate with APIs. This includes validating inputs, authenticating requests, and ensuring data confidentiality and integrity during transmission. The goal is to protect the application and its data from common API-related vulnerabilities such as injection attacks, unauthorized access, and data exposure. |

| **Noncompliant Code** |
| --- |
| This code directly concatenates user input (**userId**) into the API endpoint without any form of validation or sanitation. This practice is insecure because it assumes the input is safe, which might not always be the case. An attacker could exploit this by manipulating the **userId** input to access or manipulate data belonging to other users, leading to an Insecure Direct Object Reference (IDOR) vulnerability. |
| // Noncompliant: Insecure API call without validation  std::string fetchUserData(std::string userId) {  return apiCall("/user/data/" + userId); // Potential IDOR vulnerability  } |

| **Compliant Code** |
| --- |
| In contrast, the compliant code includes a validation step (**isValidUserId(userId)**) before using the **userId** in the API call. This check ensures that the input adheres to expected criteria (such as format, length, or value range) and is deemed safe for use. By validating the input first, the code mitigates the risk of IDOR and other related vulnerabilities, ensuring that only authorized and validated requests are processed. This approach exemplifies a proactive stance on security, where inputs are treated as untrusted by default and must be explicitly verified before use. |
| // Compliant: Secure API call with input validation  std::string fetchUserData(std::string userId) {  if (!isValidUserId(userId)) {  throw std::invalid\_argument("Invalid User ID");  }  return apiCall("/user/data/" + userId); // User ID is validated  } |

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

| **Principles(s):** “Validate Input Data" principle is applied here to ensure that the API receives a valid and expected user ID, preventing potential insecure direct object references (IDOR) and other API-related vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Moderate | Medium - | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| API Security Scanner | 2.3 | APISecurityCheck | Scans API calls for security vulnerabilities like IDOR. |

**Coding Standard 12**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Resource Access Controls** | STD-012-CPP | "Resource Access Controls" refers to the mechanisms and policies put in place to ensure that users can only access or modify resources, such as files or data, for which they have explicit permission. This standard is critical for enforcing the principle of least privilege and preventing unauthorized access or manipulation of sensitive information, thereby safeguarding the integrity and confidentiality of the system's resources. |

| **Noncompliant Code** |
| --- |
| The non-compliant code in the context of resource access controls shows a file deletion operation that proceeds without verifying the user's permissions. This oversight means any user, regardless of their access level, could potentially delete files, leading to unauthorized data manipulation or loss. The absence of an access control check undermines the principle of least privilege, making the system vulnerable to insider threats or privilege escalation attacks. |
| // Noncompliant: No access control check  void deleteFile(std::string filePath) {  // Code to delete file without checking user's permission  } |

| **Compliant Code** |
| --- |
| In the compliant code, before a file deletion operation is performed (**deleteFile** function), it checks if the user has the necessary permissions for the action using **user.hasPermission(filePath, Action::Delete)**. This ensures that users can only perform actions they are explicitly authorized to do, adhering to the Principle of Least Privilege and effectively securing resources against unauthorized access. |
| // Compliant: Enforcing access control  void deleteFile(std::string filePath, User user) {  if (!user.hasPermission(filePath, Action::Delete)) {  throw std::invalid\_argument("Permission Denied");  }  // Code to delete file with proper permission check  } |

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

| **Principles(s):** The principle of "Least Privilege" is used to ensure users can only perform actions they're permitted to, preventing unauthorized access. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Moderate | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Access Control Tester | 1.1 | AccessControlCheck | Checks code to verify proper access controls are in place for resources. |

#### Coding Standard 13

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Secure Session Management** | STD-013-CPP | "Secure Session Management" focuses on the secure handling of user sessions within an application. It covers aspects such as the generation, storage, and validation of session tokens, ensuring that sessions are properly established and terminated, and protecting against session-related attacks like session hijacking or fixation. Proper session management is crucial for maintaining user state and authentication status securely across multiple requests. |

| **Noncompliant Code** |
| --- |
| In the non-compliant code example for secure session management, the session initiation process (**startSession** function) blindly trusts and uses the provided **userToken** without any form of validation. This approach is insecure as it does not verify the authenticity or integrity of the session token, leaving the application vulnerable to session hijacking or fixation attacks, where an attacker could use a forged or stolen session token to gain unauthorized access to user sessions. |
| // Noncompliant: Insecure session token handling  void startSession(std::string userToken) {  // Directly trusts and uses the userToken without verification  } |

| **Compliant Code** |
| --- |
| The compliant code showcases a secure approach to session management by validating the session token (**userToken**) through **validateToken(userToken)**. This validation confirms the legitimacy of the session token before initiating a user session. By incorporating this validation step, the code prevents unauthorized session access, ensuring that sessions are securely managed and protected against potential session hijacking or token manipulation attacks. |
| // Compliant: Secure session token validation  void startSession(std::string userToken) {  if (!validateToken(userToken)) {  throw std::invalid\_argument("Invalid Session Token");  }  // Code to handle session with a valid token  } |

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

| **Principles(s):** "Validate Input Data" principle ensures that session tokens are verified, which is crucial for secure session management. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | High - | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Session Security Tool | 1.5 | SessionSecurityCheck | Automates the validation of session tokens to prevent unauthorized access. |

#### Coding Standard 14

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Concurrency**  Secure Data Serialization and Deserialization | STD-014-CPP | Implementing Secure Practices in Data Serialization and Deserialization |

| **Noncompliant Code** |
| --- |
| The non-compliant code demonstrates an unsafe practice of deserializing user-supplied data without any security checks. This lack of validation and sanitation makes the code vulnerable to attacks such as remote code execution or data tampering. Attackers could exploit this by crafting malicious serialized data that, when deserialized, could execute arbitrary code or alter application state in unintended ways. The absence of safeguards in the deserialization process exposes the application to significant security risks. |
| // Noncompliant: Unsafe deserialization of user-supplied data  Object deserializeUserData(std::string serializedData) {  // Code to deserialize data without any security checks  } |

| **Compliant Code** |
| --- |
| The compliant code includes a validation step (**isValidSerializedData**) that checks the format and content of the serialized data before deserialization. This mitigates risks associated with deserializing malicious or malformed data, such as remote code execution or data tampering. |
| // Compliant: Secure deserialization with checks against unexpected data types  Object deserializeUserData(std::string serializedData) {  if (!isValidSerializedData(serializedData)) {  throw std::invalid\_argument("Invalid serialized data format");  }  // Proceed with deserialization only after validation  return safeDeserialize(serializedData); // Use a secure method for deserialization  } |

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

| **Principles(s):** Insecure deserialization can lead to severe vulnerabilities like remote code execution, making it critical to address with high priority. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Secure Deserialization Tool | 1.0 | SecureDeserializationCheck | Scans for unsafe deserialization practices and suggests secure alternatives. |

### 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.

Automation in the context of DevSecOps is about integrating security as a core part of the development and operations process. Using the provided DevSecOps diagram, automation should be applied at various stages of the cycle to enforce and comply with the defined security standards.

1. **Design**: Security test-driven design should be automated to ensure that security considerations are part of the initial build. This includes the integration of security-focused unit tests and the automation of code analysis tools that can detect vulnerabilities early.
2. **Build**: The build process should automatically include security checks, such as dependency checks for known vulnerabilities. Tools like OWASP Dependency-Check can be used here.
3. **Verify and Test**: Automated vulnerability scanning, and security testing should be part of the continuous integration pipeline. Upon each commit, code changes should trigger a series of automated security tests along with functional tests.
4. **Pre-production**: Before deployment, automated penetration testing tools can simulate attacks on the application to find any last-minute vulnerabilities.
5. **Transition and Health Check**: Automate the configuration and security settings of the deployment environment, ensuring that they meet predefined security standards.
6. **Production**: In production, automation should include monitoring tools that provide real-time alerts for any security issues. Additionally, automate the response to certain types of incidents, such as blocking IP addresses that are sources of an attack.
7. **Maintain and Stabilize**: Automated processes should compare current configurations against security baselines, ensuring that any drift is corrected, and that the system returns to a secure state after any incident.
8. **Monitor and Detect**: Automate the logging and analysis of security events using tools like SIEM (Security Information and Event Management), which can correlate data and identify potential security incidents.
9. **Respond**: Automating the response to detected threats is critical. This can include shutting down services, isolating affected systems, and—if necessary—triggering a rollback to a secure state.

By automating these aspects of the DevSecOps cycle, Green Pace ensures that security is not just a manual checklist item but a robust, integrated part of the entire development and operations workflow. This will minimize human error, provide rapid feedback on security issues, and maintain a strong security posture across all phases of application delivery.

### 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 | High | Low | High | 4 |
| STD-003-CPP | High | Moderate | Medium | High | 4 |
| STD-004-CPP | Critical | High | Low to Medium | High | 5 |
| STD-005-CPP | High | Moderate | Medium | High | 4 |
| STD-006-CPP | Medium | High | Low | High | 3 |
| STD-007-CPP | Medium | Moderate | Medium | High | 4 |
| STD-008-CPP | High | High | Medium | High | 5 |
| STD-009-CPP | High | High | Medium | High | 5 |
| STD-010-CPP | High | Moderate | Medium | High | 4 |
| STD-011-CPP | High | Moderate | Medium | High | 4 |
| STD-012-CPP | Medium | Moderate | Low | Medium | 3 |
| STD-013-CPP | High | High | High | High | 5 |
| STD-014-CPP | Critical | High | High | High | 5 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | **Policy**: All sensitive data stored on disk, including databases, file systems, and backups, must be encrypted using industry-standard encryption algorithms.  **Application**: This policy applies to all data storage systems within the organization.  **Rationale**: Encryption at rest prevents unauthorized access to data in the event of physical theft or unauthorized access to the physical storage. |
| Encryption in flight | **Policy**: Any data transmitted over a network must be encrypted using secure communication protocols such as TLS.  **Application**: This policy is applicable for all data exchanges across internal and external networks.  **Rationale**: Encryption in flight protects data from being intercepted and read by unauthorized parties during transmission. |
| Encryption in use | **Policy**: Data being processed in memory should be encrypted where feasible, especially when handling sensitive information.  **Application**: This applies to data temporarily loaded into memory or cache during normal application operation.  **Rationale**: Protects sensitive information from being exposed in system memory, potentially accessible through memory dumps or system exploits. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Policy**: Systems must authenticate users' identities via secure methods like passwords, biometrics, or multi-factor authentication before granting access.  **Application**: Mandatory for user logins, addition of new users, and any identity verification processes.  **Rationale**: Authentication ensures that users are who they claim to be and prevents unauthorized access. |
| Authorization | **Policy**: Once authenticated, users must be authorized to access only the resources appropriate for their roles.  **Application**: Applicable to changes to the database, user levels of access, and file access. **Rationale**: Authorization ensures users can only perform actions allowed by their permissions, enforcing the principle of least privilege. |
| Accounting | **Policy**: All user actions must be logged and audited to provide an accountable record of operations.  **Application**: Relevant for user logins, changes to the database, file accesses, and system changes.  **Rationale**: Accounting provides a trail of user actions for auditing, assisting in understanding the scope of any breach and identifying malicious activities. |

**\***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

Here's a mapping of the principles to each of the 14 coding standards, with justification for each connection:

**STD-001-CPP: Strong Type Checking**

* Principles 4 and 10: Strong type checking simplifies the code and aligns with established secure coding standards by preventing type-related errors.

**STD-002-CPP: Validate Data Range**

* Principle 1: Ensures that all data adheres to the expected range, thereby preventing errors and potential security vulnerabilities.

**STD-003-CPP: Safe String Handling**

* Principles 4 and 8: Keeping string handling simple and using secure functions align with defense in depth, reducing the chance of vulnerabilities like buffer overflows.

**STD-004-CPP: Preventing SQL Injection**

* Principles 1 and 8: Input validation is crucial to prevent SQL injection, a key part of the defense in depth strategy against a variety of injection-based attacks.

**STD-005-CPP: Secure Memory Management**

* Principle 8: Ensuring memory is allocated, used, and freed correctly adds a layer of security, protecting against vulnerabilities like buffer overflows.

**STD-006-CPP: Proper Use of Assertions**

* Principle 4: Using assertions properly during development keeps the error handling simple and avoids misusing them for runtime error handling in production.

**STD-007-CPP: Secure Exception Handling**

* Principle 6: Handling exceptions at the appropriate level without exposing sensitive information ensures the principle of least privilege is maintained.

**STD-008-CPP: Comprehensive Input Validation**

* Principle 1: Thorough input validation at every entry point of the application is essential for preventing many types of common vulnerabilities.

**STD-009-CPP: Secure Cryptographic Practices**

* Principles 8 and 10: Using strong, up-to-date cryptographic methods and key management is a crucial aspect of a defense in depth strategy and adheres to secure coding standards.

**STD-010-CPP: Secure Concurrency Management**

* Principles 4 and 8: Simple, thread-safe functions and proper synchronization prevent concurrency issues, adding security layers to the application.

**STD-011-CPP: Secure API Interaction**

* Principles 1 and 8: Validating all data sent to and received from APIs prevents threats like IDOR, supported by a defense in depth approach.

**STD-012-CPP: Resource Access Controls**

* Principles 5 and 6: Implementing access controls that default to deny unless explicitly granted, following the principle of least privilege for data access.

**STD-013-CPP: Secure Session Management**

* Principles 3, 6, and 9: Secure session management should be architected into the system from the beginning, ensuring only necessary access and comprehensive quality assurance.

**STD-014-CPP: Secure Data Serialization and Deserialization**

* Principles 4, 8, and 10: Implementing secure practices for serialization/deserialization aligns with keeping the code simple, defense in depth, and established secure coding standards.

By linking these principles to the coding standards, we show a deliberate and strategic approach to security that integrates best practices throughout the software development lifecycle. This linkage not only fortifies the code against potential threats but also ensures that the security measures are understandable, manageable, and verifiable.

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 | 03/23/2024 | Module Three Milestone | Steven Stutts | [Insert text.] |
| 1.2 | 04/08/2024 | Project One | Steven Stutts | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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