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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Scope

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

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input validation is an important security principle that ensures all data entering a system is checked for correctness, type, length, and format. By validating input, developers can prevent malicious data from causing harm, such as injection attacks, buffer overflows, or unexpected behavior. For example, if a user submits a form, the system should verify that the input matches the expected format (e.g., an email address should contain an "@" symbol). Proper input validation reduces the attack surface and ensures that only safe, well-formed data is processed by the application. |
| 1. Heed Compiler Warnings | Compiler warnings are often indicators of issues in the code, such as uninitialized variables, type mismatches, or unsafe practices. Ignoring these warnings can lead to vulnerabilities that attackers might exploit. By addressing compiler warnings, developers can catch and fix problems early in the development process, reducing the likelihood of security flaws. For example, a warning about an uninitialized variable could lead to undefined behavior, which might be exploited to crash the program or execute arbitrary code. |
| 1. Architect and Design for Security Policies | Security should be integrated into the architecture and design of a system from the very beginning, rather than being added as an afterthought. This principle involves identifying potential threats during the design phase and implementing controls to mitigate them. For example, a secure architecture might include layers of defense, such as firewalls, encryption, and access controls, to protect sensitive data. By designing with security in mind, developers can create systems that are resistant to many attacks and easier to maintain over time. |
| 1. Keep It Simple | Complex systems are harder to secure because they have more potential points of failure and are more difficult to understand and maintain. The principle of keeping it simple encourages developers to avoid unnecessary complexity in their designs and implementations. Simple systems are easier to audit, test, and secure. For example, using straightforward algorithms and avoiding overly convoluted logic can reduce the risk of introducing vulnerabilities. Simplicity also makes it easier for other developers to review and understand the code, which is essential for maintaining security. |
| 1. Default Deny | The default deny principle states that access to resources should be denied by default unless explicitly granted. This minimizes the attack surface by ensuring that only authorized users or processes can access sensitive data or functionality. For example, a firewall should block all incoming traffic by default, allowing only specific, trusted connections. Similarly, user accounts should have no permissions by default, with access granted only as needed. This approach reduces the risk of accidental exposure or unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege (PoLP) dictates that users, processes, and systems should have the minimum level of access necessary to perform their functions. By limiting access, developers can reduce the potential damage caused by compromised accounts or malicious actors. For example, a database user should only have access to the specific tables and operations required for their role, rather than full administrative privileges. This principle helps contain breaches and limits the spread of attacks within a system. |
| 1. Sanitize Data Sent to Other Systems | Data sent to external systems, such as databases, APIs, or third-party services, must be sanitized to prevent injection attacks or other vulnerabilities. Sanitization involves removing or escaping potentially harmful characters or sequences that could be interpreted as commands. For example, when sending data to a database, developers should use parameterized queries to prevent SQL injection. Similarly, when sending data to a web browser, output should be encoded to prevent cross-site scripting (XSS) attacks. Proper sanitization ensures that data is safe to process and display. |
| 1. Practice Defense in Depth | Defense in depth is a strategy that involves implementing multiple layers of security controls to protect against different types of attacks. If one layer fails, others can still provide protection. For example, a web application might use firewalls, intrusion detection systems, encryption, and access controls to secure its data. This approach ensures that even if an attacker bypasses one security measure, they will still face additional barriers. Defense in depth is essential for creating resilient systems that can withstand sophisticated attacks. |
| 1. Use Effective Quality Assurance Techniques | QA techniques, such as code reviews, static analysis, dynamic analysis, and penetration testing, are essential for identifying and fixing security vulnerabilities. Regular testing helps ensure that the code adheres to security standards and is free from common vulnerabilities. For example, static analysis tools can detect issues like buffer overflows or insecure API usage, while penetration testing can simulate real-world attacks to identify weaknesses. By incorporating QA into the development process, teams can catch and address security issues before they reach production. |
| 1. Adopt a Secure Coding Standard | A secure coding standard provides developers with a set of guidelines and best practices for writing secure code. By following a standard, such as the SEI CERT C++ Coding Standard, developers can avoid common pitfalls and ensure consistency across the codebase. For example, a secure coding standard might require the use of safe string functions, proper error handling, and input validation. Adopting a secure coding standard helps create a culture of security within the development team and reduces the likelihood of introducing vulnerabilities. |

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Strong data types help prevent type confusion and ensure that variables are used in a way that aligns with their intended purpose. Using the correct data types reduces the risk of vulnerabilities such as buffer overflows, integer overflows, and type mismatches. For example, using `std::array` instead of raw arrays ensures bounds checking, which prevents out-of-bounds access. Strong typing also makes the code more readable and maintainable, as the intent of each variable is clear. |

| **Noncompliant Code** |
| --- |
| This code demonstrates an unsafe implicit type conversion that attempts to store a large integer value into a smaller char type, which results in data truncation and potentially unpredictable output. |
| int largeValue = 900;  char smallValue = largeValue; *// Data loss due to truncation*  std::cout << (int)smallValue; *// Unpredictable output* |

| **Compliant Code** |
| --- |
| The code shows a safe approach to type conversion by using static\_cast to explicitly convert the integer to a 16-bit type, preserving data integrity and preventing unexpected behavior. |
| int largeValue = 900; std::int16\_t safeValue = static\_cast<std::int16\_t>(largeValue); *// Explicit type conversion* std::cout << safeValue; *// Preserves data integrity* |

| **Principles:**  **(2) Heed Compiler Warnings** *Reasoning:* Strong typing often triggers compiler warnings if used incorrectly. By paying attention to these warnings, developers can prevent subtle type-conversion errors.  **(10) Adopt a Secure Coding Standard** *Reasoning:* This standard directly aligns with recommended SEI CERT guidelines for type safety and helps ensure consistent, secure handling of data types across the codebase. |
| --- |

**Threat Level**

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

***Explanation:*** Type confusion or unsafe casts can lead to memory corruption or logic errors. These issues are relatively easy to fix once identified, hence “Low” remediation cost, but the impact can be severe.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | cppcoreguidelines-pro-type-reinterpret-cast (example) | Detects potentially unsafe type conversions or reinterpret casts that could lead to data corruption or undefined behavior. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Input validation is essential to ensure that data entering the system is safe and conforms to expected formats. Without proper validation, malicious input can lead to vulnerabilities such as injection attacks, buffer overflows, or unexpected behavior. For example, validating user input for length, type, and format can prevent SQL injection or cross-site scripting (XSS) attacks. Input validation should occur as close to the source as possible to minimize the risk of malicious data propagating through the system. |

| **Noncompliant Code** |
| --- |
| This example illustrates a dangerous method of directly inserting user input into a SQL query, which leaves the code vulnerable to SQL injection attacks. |
| void processUserInput(std::string input) {  *//No input validation*  database.execute("SELECT \* FROM users WHERE username = '" + input + "'"); } |

| **Compliant Code** |
| --- |
| The code implements robust input validation by checking the input's length and content, and uses parameterized queries to prevent potential SQL injection vulnerabilities. |
| void processUserInput(const std::string& input) {  *// Validate input length and content* if (input.empty() || input.length() > 50 || !isValidUsername(input)) {  throw std::invalid\_argument("Invalid username");  } *// Use parameterized query to prevent SQL injection* preparedStatement.setString(1, input);  } |

| **Principles(s):**  **(1) Validate Input Data** *Reasoning:* This standard directly enforces validation of data to prevent malformed or malicious input from entering the system.  **(7) Sanitize Data Sent to Other Systems** *Reasoning:* Proper input validation is the first step before sanitizing data that will be passed to external services (e.g., databases, APIs). |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 4 |

***Explanation:*** Improper input validation is one of the most common and dangerous security flaws (e.g., injection attacks). Fixing input validation can be straightforward (low cost), but the impact of an exploit can be severe.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 21.2 | Dynamic Code Injection – Input Validation (example) | Identifies places where unvalidated or improperly validated input could be used in a way that leads to injection attacks. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | String manipulation is a common source of vulnerabilities, such as buffer overflows and null-termination errors. Using safe string functions ensures that strings are handled securely, preventing these issues. For example, functions like strncpy and std::string in C++ provide built-in protections against buffer overflows. Safe string handling also includes ensuring proper null termination and avoiding the use of deprecated or unsafe functions like strcpy. |

| **Noncompliant Code** |
| --- |
| This code uses the unsafe strcpy() function, which can cause a buffer overflow by copying a string that exceeds the buffer's allocated size. |
| char buffer[10];  strcpy(buffer, "This is a very long string that will cause buffer overflow"); |

| **Compliant Code** |
| --- |
| The implementation uses safer string handling techniques, including std::string, std::array, and strncpy() with explicit null-termination to prevent buffer overflow risks. |
| std::string safeString = "Safely managed string";  std::array<char, 50> buffer{};  std::strncpy(buffer.data(), safeString.c\_str(), buffer.size() - 1); buffer[buffer.size() - 1] = '\0'; *// Ensure null-termination* |

| **Principles:**  **(9) Use Effective Quality Assurance Techniques** *Reasoning:* Safe string handling often involves using static/dynamic analysis tools to detect buffer overflows or unsafe string functions.  **(10) Adopt a Secure Coding Standard** *Reasoning:* String correctness is a major focus in SEI CERT guidelines to avoid buffer overflows and ensure null termination. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 2 |

***Explanation:*** String-related bugs can be very damaging (potential buffer overflows), but they’re also typically easy to detect and fix with the right tooling and coding practices.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.x | “C++ S3514: Use Safe String Functions” (example) | Checks for unsafe string operations (e.g., strcpy, sprintf) and suggests safer alternatives or usage of std::string. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL injection is a critical vulnerability that occurs when user input is directly included in SQL queries, allowing attackers to execute arbitrary SQL commands. Parameterized queries separate SQL code from user input, preventing injection attacks. By using prepared statements or parameterized queries, developers can ensure that user input is treated as data, not executable code. This practice is essential for securing database interactions. |

| **Noncompliant Code** |
| --- |
| The code demonstrates an insecure method of constructing SQL queries by directly concatenating user input, which opens the door to potential SQL injection attacks. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  database.execute(query); *// Vulnerable to SQL injection* |

| **Compliant Code** |
| --- |
| This example shows proper database interaction using prepared statements and parameterized queries, effectively preventing SQL injection vulnerabilities. |
| std::string username = getUserInput();  PreparedStatement stmt = database.prepare("SELECT \* FROM users WHERE username = ?");  stmt.setString(1, username);  ResultSet result = stmt.executeQuery(); *// Parameterized query prevents injection* |

| **Principles:**  **(1) Validate Input Data** *Reasoning:* Validating user-supplied input before using it in SQL queries is the first line of defense.  **(7) Sanitize Data Sent to Other Systems** *Reasoning:* All data going to the database must be sanitized/parameterized to prevent injection.  **(3) Architect and Design for Security Policies** *Reasoning:* Database interactions should be designed with security in mind from the start, using parameterized queries or stored procedures. |
| --- |

**Threat Level**

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

***Explanation:*** SQL injection is one of the most critical vulnerabilities (OWASP Top 10). Mitigation requires code changes and possibly redesign of database interaction patterns.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.x | “C++ S5889: Potential SQL Injection” | Flags any instance where strings are concatenated into SQL queries and recommends parameterized queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Manual memory management is error-prone and can lead to vulnerabilities such as memory leaks, dangling pointers, and double-free errors. Smart pointers, such as `std::unique\_ptr` and `std::shared\_ptr`, automatically manage memory allocation and deallocation, reducing the risk of these issues. By using smart pointers, developers can ensure that memory is properly managed, even in the presence of exceptions or early returns. |

| **Noncompliant Code** |
| --- |
| The code manually manages memory allocation with new and delete, which can lead to memory leaks and improper resource management. |
| int\* ptr = new int(42); *// No guarantee of proper deletion, potential memory leak*  delete ptr;  ptr = nullptr; |

| **Compliant Code** |
| --- |
| By using std::unique\_ptr, the code ensures automatic and safe memory management, reducing the risk of memory-related errors. |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(42);  *// Automatic memory management, no manual deletion needed* |

| **Principles(s):**  **(8) Practice Defense in Depth** *Reasoning:* Proper memory management (e.g., smart pointers) adds a layer of protection against memory corruption, complementing other security controls.  **(9) Use Effective Quality Assurance Techniques** *Reasoning:* Tools like AddressSanitizer, Valgrind, or static analyzers can detect memory leaks and invalid accesses early. |
| --- |

**Threat Level**

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

***Explanation:*** Memory-related bugs can lead to severe vulnerabilities (e.g., arbitrary code execution). They can be harder to fix if the codebase is large and uses manual memory management extensively.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer (part of Clang/LLVM) | Included with Clang 14+ | ASan runtime checks | Dynamically detects memory errors (out-of-bounds, use-after-free) during testing, helping to pinpoint dangerous code paths. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are useful for catching logical errors during development, but they should not be used for runtime error handling. Assertions are typically removed in production builds, so relying on them for runtime checks can lead to vulnerabilities. Instead, developers should use proper error handling mechanisms, such as exceptions or return codes, to handle runtime errors. Assertions should only be used to verify assumptions that should always hold true during development. |

| **Noncompliant Code** |
| --- |
| This code relies on an assertion that will be removed in release builds, making it an unreliable method for runtime error checking. |
| void processData(std::vector<int>& data) { assert(!data.empty());  *// Removed in release build, unsafe for runtime check*  *// Process data*  } |

| **Compliant Code** |
| --- |
| The implementation provides proper error handling with runtime checks and exception throwing, ensuring robust error management across all build configurations. |
| void processData(const std::vector<int>& data) { if (data.empty()) { throw std::invalid\_argument("Data vector cannot be empty");  }  *// Process data with proper error handling* } |

| **Principles(s):**  **(2) Heed Compiler Warnings** *Reasoning:* Over-reliance on assertions can mask actual runtime checks, which might generate warnings or errors if used improperly.  **(9) Use Effective Quality Assurance Techniques** *Reasoning:* QA practices should ensure that runtime error handling (not just assert) is tested thoroughly in production builds. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 2 |

***Explanation:*** Using assertions in place of proper error handling can lead to vulnerabilities if those assertions are disabled in production.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | *missingErrorCheck* (example) | Detects places where error conditions are not properly handled (e.g., only using assert), encouraging robust runtime checks. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling ensures that the program can recover from unexpected errors and continue operating safely. Catching specific exceptions, rather than using catch-all handlers, allows developers to handle errors appropriately and provide meaningful feedback to users. For example, catching a std::out\_of\_range exception allows the program to handle array index errors gracefully, rather than crashing. Exception handling is essential for creating robust and secure applications. |

| **Noncompliant Code** |
| --- |
| The code attempts to open a file without implementing any error checking or exception handling mechanisms. |
| int readFile(const std::string& filename) { FILE\* file = fopen(filename.c\_str(), "r");  *// No error checking or handling* return 0;  } |

| **Compliant Code** |
| --- |
| This version implements a comprehensive try-catch block with specific exception handling for file operations, providing more robust error management. |
| int readFile(const std::string& filename) {  try {  std::ifstream file(filename);  if (!file.is\_open()) {  throw std::runtime\_error("Could not open file: " + filename);  }  *// Process file with specific exception handling* } catch (const std::exception& e) { std::cerr << "File error: " << e.what() << std::endl;  return -1;  }  } |

| **Principles(s):**  **(3) Architect and Design for Security Policies** *Reasoning:* Proper exception handling design ensures that the system fails safely and can recover gracefully.  **(9) Use Effective Quality Assurance Techniques** *Reasoning:* Code reviews and testing can confirm that exceptions are caught properly and do not reveal sensitive information or crash the system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 2 |

***Explanation:*** Poorly handled exceptions can lead to crashes or leaks of sensitive information, but adding correct exception handling is often straightforward once identified.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | cppcoreguidelines-avoid-catch-all | Warns against catch-all (catch(...)) blocks and encourages catching specific exceptions for more secure and predictable error handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Proper Error Handling | STD-008-CPP | Proper error handling ensures that the program can recover from unexpected errors and continue operating safely. This includes checking return values, using exceptions, and providing meaningful error messages. For example, checking the return value of a file operation ensures that the program can handle file access errors gracefully. Proper error handling is essential for creating robust and secure applications. |

| **Noncompliant Code** |
| --- |
| The code performs division without checking for the potential division by zero, which can cause runtime errors. |
| int divideTwoNumbers(int a, int b) {  return a / b; *// No error checking for division by zero*  } |

| **Compliant Code** |
| --- |
| By adding an explicit check for division by zero and throwing an exception, the code prevents unexpected runtime errors. |
| int divideTwoNumbers(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Division by zero");  }  return a / b;  } |

| **Principles:**  **(9) Use Effective Quality Assurance Techniques** *Reasoning:* Proper error handling must be tested thoroughly to ensure it handles edge cases.  **(4) Keep It Simple** *Reasoning:* Straightforward error-handling code is easier to maintain and audit. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 2 |

***Explanation:*** Lack of proper error handling can result in unhandled exceptions, crashes, or silent failures, but it’s typically not as severe as memory corruption or injection attacks.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.x | “C++ S2221: Return codes should be tested” | Checks whether function return codes and error states are properly handled rather than ignored. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Ensure Thread Safety | STD-009-CPP | In multi-threaded applications, thread safety is essential to prevent race conditions and ensure correct behavior. Race conditions occur when multiple threads access shared resources simultaneously without proper synchronization, leading to unpredictable results. Using synchronization mechanisms, such as mutexes or atomic operations, ensures that shared resources are accessed safely. Thread safety is important for creating reliable and secure multi-threaded applications. |

| **Noncompliant Code** |
| --- |
| The code uses a raw integer variable that can lead to race conditions in multi-threaded environments. |
| int sharedCounter = 0;  void incrementCounter() {  sharedCounter++; *// Race condition possible*  } |

| **Compliant Code** |
| --- |
| Using std::atomic ensures thread-safe incrementing of a shared counter, preventing potential synchronization issues. |
| std::atomic<int> sharedCounter{0};  void incrementCounter() {  sharedCounter++; *// Atomic operation ensures thread safety*  } |

| **Principles:**  **(8) Practice Defense in Depth** *Reasoning:* Thread safety is another layer of security, ensuring concurrency does not introduce race conditions that attackers could exploit.  **(3) Architect and Design for Security Policies** *Reasoning:* Designing multi-threaded systems with security in mind (mutexes, atomics, etc.) prevents data corruption and potential vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Medium | Medium | 2 |

***Explanation:*** Race conditions can be severe but are often less likely than input or memory bugs. Fixes sometimes require architectural changes, thus “Medium” cost.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer (part of Clang/LLVM) | Included with Clang 14+ | TSan runtime checks | Dynamically detects data races and synchronization issues during multi-threaded tests. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Secure File Handling** | STD-010-CPP | Secure file handling ensures that files are accessed and manipulated safely, preventing unauthorized access or data corruption. This includes checking file permissions, validating file paths, and handling file operations securely. For example, validating file paths prevents directory traversal attacks, while checking file permissions ensures that only authorized users can access sensitive files. Secure file handling is essential for protecting sensitive data and maintaining system integrity. |

| **Noncompliant Code** |
| --- |
| This code opens a file without validating the file path or checking necessary permissions. |
| void processFile(const std::string& filepath) {  std::ofstream file(filepath);  *// No permission or path validation*  } |

| **Compliant Code** |
| --- |
| The implementation adds important validation for file path and permissions before performing file operations, enhancing overall file handling security. |
| void processFile(const std::string& filepath) {  *// Validate file path to prevent directory traversal*  if (!isValidAndSafePath(filepath)) {  throw std::invalid\_argument("Unsafe file path");  }  *// Check file permissions*  if (!hasWritePermission(filepath)) {  throw std::runtime\_error("Insufficient file permissions");  }  std::ofstream file(filepath);  } |

| **Principles:**  **(5) Default Deny** *Reasoning:* When handling files, the default posture should be to deny access unless explicitly allowed, minimizing risk of unauthorized file access.  **(6) Adhere to the Principle of Least Privilege** *Reasoning:* Processes and users should only have the minimum file permissions necessary, reducing potential impact if compromised. |
| --- |

**Threat Level**

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

***Explanation:*** Improper file handling can expose sensitive data or allow attackers to write files in restricted locations. Although less likely than user-input vulnerabilities, the impact can be significant.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 21.1 | Path Manipulation / File Permission Checks | Identifies places where file paths are not validated or where file operations may exceed the intended permissions. |

### Defense-in-Depth Illustration

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





**Automation Explanation**

Green Pace builds security into every stage of its DevOps process to ensure our coding standards are met. During planning, developers add security requirements and threat modeling to their user stories, and we set up automated scans so that new code is checked immediately after it is committed. As code is written, developers use simple IDE plugins (like SonarLint or Visual Studio Code extensions) that give real-time feedback on potential issues such as unsafe type conversions or weak input validation. This early feedback helps stop problems from spreading in the code.

When code is pushed to our shared repository, our automated builds run static security tests (using tools like Clang-Tidy, Cppcheck, or Fortify SCA) to check for issues with memory safety, input validation, and file handling. We also run software composition analysis to spot known vulnerabilities in third-party libraries. Before releasing the final product, we run dynamic tests like fuzz testing and integration tests to catch runtime errors that static tests might miss. Tools such as AddressSanitizer or ThreadSanitizer help us detect memory leaks or concurrency problems. Only when all these tests pass do we sign the final binaries, and any critical vulnerabilities block a release until they are fixed.

In production, we keep a close watch on our applications using runtime self-protection and regular penetration testing to find any new or missed issues. If a vulnerability or incident is detected, we quickly isolate the affected systems and work with the development team to resolve the issue. We also use threat intelligence feeds to update our scanning rules and security policies. By continuously planning, testing, monitoring, and adapting, we ensure that security remains a core part of our DevOps workflow, reducing both the risk and cost of fixing vulnerabilities while maintaining a strong defense.

### Summary of Risk Assessments

| 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 | Medium | Medium | Low | Medium | 2 |
| STD-004-CPP | High | High | Medium | High | 5 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 2 |
| STD-007-CPP | Medium | Medium | Low | Medium | 2 |
| STD-008-CPP | Medium | Medium | Low | Medium | 2 |
| STD-009-CPP | Medium | Low | Medium | Medium | 2 |
| STD-010-CPP | High | Low | Medium | Medium | 3 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest means protecting data stored on disks, databases, or backup media by converting it into a form that cannot be read without the proper decryption key. This policy applies to all sensitive data within our systems. By using standard algorithms like AES-256, even if physical media are lost or stolen, the data remains secure from unauthorized access. In practice, any system storing confidential or sensitive information must use strong encryption at rest to ensure data integrity and confidentiality at all times. |
| Encryption in flight | Encryption in flight is about securing data as it moves across networks. This policy covers every communication channel that carries sensitive information, whether between internal servers, from a client to a server, or over public networks. Data must be encrypted using secure protocols such as TLS/SSL to prevent interception, eavesdropping, or man-in-the-middle attacks. This ensures that information remains private and intact during transmission between systems and users. |
| Encryption in use | Encryption in use protects data while it is actively being processed by an application. This is especially important in environments where data must stay confidential even as it’s used. Techniques like secure enclaves, hardware-based memory encryption, or homomorphic encryption help ensure that data is not exposed in plain text during processing. This policy is critical for systems that handle highly sensitive data in real time, ensuring both privacy and performance. |

| 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 anyone trying to access our systems or data. This policy requires the use of strong methods, such as multi-factor authentication, for all user logins and access points. By ensuring that every user is properly verified, we reduce the risk of unauthorized access and protect against weak or stolen credentials. |
| Authorization | Authorization controls who can access what resources, based on a user’s verified identity and assigned privileges. This policy enforces the principle of least privilege, meaning that users only have access to the information and functions necessary for their roles. It covers managing permissions for tasks like changing database records, adding new users, and setting user access levels. Strict control over access helps prevent accidental or intentional misuse of system resources. |
| Accounting | Accounting, or auditing, involves tracking and logging user activities and changes in our systems. This policy requires that every important action such as user logins, changes to the database, addition of new users, modifications of user access levels, and file accesses be recorded in audit logs. Regular review of these logs helps detect any unusual activities, supports compliance, and aids in investigations if a security incident occurs. Maintaining detailed logs ensures that any unauthorized actions are quickly identified and addressed. |

## 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 | 02/15/2025 | Updated policy to add new encryption and Triple-A sections, simplified automation language, and consolidated risk assessments. | Alexander Wagner | [CISO name] |
| [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 |