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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All input data from users, external systems, or any untrusted source must be thoroughly validated before processing. This includes checking data types, ranges, formats, and lengths to ensure they meet expected criteria. Input validation prevents malicious data from entering the system and protects against attacks like SQL injection, buffer overflows, and cross-site scripting. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential security vulnerabilities, coding errors, or unsafe practices that could lead to exploitable conditions. Modern compilers are sophisticated tools that can detect issues like buffer overflows, uninitialized variables, type mismatches, and deprecated functions. Treating warnings as errors and addressing them promptly helps identify security flaws early in the development process. |
| 1. Architect and Design for Security Policies | Security must be integrated into the system architecture and design from the beginning, not added as an afterthought. This involves identifying security requirements, threat models, and implementing appropriate security controls at the architectural level. Security policies should guide design decisions about data flow, access controls, authentication mechanisms, and system boundaries. |
| 1. Keep It Simple | Complex systems are more difficult to secure, analyze, and maintain, creating more opportunities for security vulnerabilities. Simple designs are easier to understand, test, and verify, reducing the likelihood of implementation errors that could lead to security flaws. This principle advocates for minimizing unnecessary features, reducing code complexity, and choosing straightforward solutions over elaborate ones. |
| 1. Default Deny | Access control systems should deny access by default and only grant permissions when explicitly authorized. This principle ensures that if access control mechanisms fail or are bypassed, the system remains secure because no access is granted without explicit permission. Rather than starting with open access and trying to restrict it, systems should begin with no access and carefully grant only the minimum necessary permissions. |
| 1. Adhere to the Principle of Least Privilege | Users, processes, and systems should be granted only the minimum level of access necessary to perform their required functions. This limits the potential damage from compromised accounts, malicious insiders, or software vulnerabilities by restricting what actions can be performed. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to external systems, databases, or other applications, it must be properly sanitized to prevent injection attacks and ensure data integrity. This involves encoding special characters, escaping potentially dangerous content, and formatting data according to the receiving system's requirements. Sanitization prevents attacks like SQL injection, command injection, and cross-site scripting by ensuring that data is treated as data rather than executable code. |
| 1. Practice Defense in Depth | Security should be implemented in multiple layers rather than relying on a single security control. This approach ensures that if one security measure fails, others are still in place to protect the system. Defense in depth involves implementing security controls at different levels such as network, host, application, and data layers. Multiple overlapping security measures create redundancy and make it much more difficult for attackers to completely compromise a system. |
| 1. Use Effective Quality Assurance Techniques | Rigorous testing and quality assurance processes are essential for identifying security vulnerabilities before software is deployed. This includes security testing, penetration testing, code reviews, static analysis, and dynamic analysis. Quality assurance should incorporate security considerations throughout the development lifecycle, not just at the end. |
| 1. Adopt a Secure Coding Standard | Development teams should follow established secure coding practices and standards that provide guidelines for writing secure code. These standards address common security vulnerabilities, specify secure coding techniques, and provide best practices for handling sensitive data, authentication, and authorization. Secure coding standards help ensure consistency across development teams and reduce the likelihood of introducing security 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 | Proper data type usage prevents undefined behavior, buffer overflows, and integer overflow vulnerabilities. Using appropriate data types with defined ranges and behaviors ensures predictable program execution and prevents security exploits that arise from type confusion or implicit conversions. |

| **Noncompliant Code** |
| --- |
| Using signed integers for array indices and size calculations can lead to negative index vulnerabilities. |
| int size = getUserInput(); // Could be negative  char buffer[100];  for (int i = 0; i < size; i++) {  buffer[i] = 'A'; // Potential buffer overflow if size > 100 or negative wraparound  } |

| **Compliant Code** |
| --- |
| Use unsigned types for sizes and indices, with proper bounds checking. |
| size\_t size = static\_cast(getUserInput());  char buffer[100];  if (size <= sizeof(buffer)) {  for (size\_t i = 0; i < size; i++) {  buffer[i] = 'A';  }  } |
|  |

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

| **Principles(s):** Adhere to the Principle of Least Privilege: Choosing the correct data type (e.g., size\_t) restricts operations to safe, intended ranges by default. It prevents unintended access to memory outside valid bounds. Proper bounds checking ensures the program only operates within safe parameters, reinforcing predictable behavior and minimizing attack possibilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | low | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | Cppcoreguidelines-\* | Analyzes C++ code against core guidelines; flags improper type usage, unsafe conversions, and missing bounds checks. |
| Cppcheck | 2.13.0 | --enable=warning | Static analysis tool that detects type mismatches, potential overflows, and array index errors. |
| SonarQube | 10.2 | C++ security rules | Identifies unsafe type handling, possible overflows, and violations of secure coding standards during continuous analysis. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validating data values before use prevents injection attacks, buffer overflows, and logic errors. Proper range checking and validation ensure that data remains within expected bounds and meets security requirements before processing. |

| **Noncompliant Code** |
| --- |
| Accepting user input without validation can lead to security vulnerabilities. |
| void processAge(int age) {  int retirement\_years = 65 - age; // No validation - could overflow  cout << "Years until retirement: " << retirement\_years << endl;  } |

| **Compliant Code** |
| --- |
| Validate input data ranges and handle invalid values appropriately. |
| bool processAge(int age) {  if (age < 0 || age > 150) {  cerr << "Invalid age value" << endl;  return false;  }  int retirement\_years = 65 - age;  cout << "Years until retirement: " << retirement\_years << endl;  return true;  } |

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

| **Principles(s):** validate input data: all user/external values should be checked against the expected format/rang/constraints before processing actually begins. By validating data values before use, the system can prevent malicious input from triggering injection attacks, buffer overflows, or unexpected logic errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | high | low | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | cppcoreguidelines-\* | Flags functions that use unvalidated or unchecked parameters and recommends adding validation logic. |
| Cppcheck | 2.13.0 | --enable=warning | Detects potential logic errors, missing boundary checks, and unsafe use of external data in variables. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Proper string handling prevents buffer overflows, null pointer dereferences, and format string attacks. Using safe string functions and proper bounds checking ensures string operations cannot be exploited to corrupt memory or execute malicious code. |

| **Noncompliant Code** |
| --- |
| Using unsafe string functions without bounds checking creates buffer overflow vulnerabilities. |
| char dest[10];  char\* src = getUserInput();  strcpy(dest, src); // No bounds checking - buffer overflow risk |

| **Compliant Code** |
| --- |
| Use safe string functions with proper size limits and validation. |
| char dest[10];  const char\* src = getUserInput();  if (src != nullptr && strlen(src) < sizeof(dest)) {  strncpy(dest, src, sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0';  } |

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

| **Principles(s):** Adopt a Secure Coding Standard: Following established secure coding practices for string handling makes sure there’s safe use of buffers and prevents vulnerabilities such as buffer overflows and format string attacks. Using safe string functions and explicit bounds checking aligns with industry standards and reduces the risk of memory corruption. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | high | low | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | hicpp-no-array-decay, cppcoreguidelines-pro-bounds-array-to-pointer-decay | Flags unsafe string operations, improper array-to-pointer decay, and missing bounds checks. |
| Cppcheck | 2.13.0 | --enable=warning | Detects unsafe string handling, buffer overflow risks, and missing null-termination safeguards. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Preventing SQL injection attacks requires proper input sanitization and the use of parameterized queries. Direct concatenation of user input into SQL statements allows attackers to manipulate database queries and potentially access or modify unauthorized data. |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries enables SQL injection attacks. |
| string username = getUserInput();  string query = "SELECT \* FROM users WHERE name = '" + username + "'";  // Vulnerable to: username = "'; DROP TABLE users; --" |

| **Compliant Code** |
| --- |
| Use parameterized queries or proper input sanitization to prevent SQL injection. |
| string username = getUserInput();  // Sanitize input - remove or escape dangerous characters  string sanitized = sanitizeInput(username);  string query = "SELECT \* FROM users WHERE name = ?";  // Use prepared statements with parameter binding |

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

| **Principles(s):** Sanitize Data to Other Systems: All data passed to external systems, including databases, must be properly sanitized or parameterized to ensure it is interpreted as data, not executable code. This prevents injection attacks such as SQL injection by neutralizing malicious input before it’s processed and reaches the database. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| critical | high | low | critical | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.2 | SQL injection security rules | Detects direct string concatenation in SQL queries and recommends parameterized queries or sanitization. |
| OWASP ZAP | 2.14.0 | Active Scan | Performs automated injection testing to identify exploitable SQL injection vulnerabilities in application code. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Proper memory management prevents buffer overflows, use-after-free vulnerabilities, and memory leaks. Following memory safety practices ensures that dynamic memory is properly allocated, used within bounds, and deallocated to prevent exploitation and system instability. |

| **Noncompliant Code** |
| --- |
| Manual memory management without proper bounds checking creates vulnerability opportunities. |
| char\* buffer = (char\*)malloc(10);  strcpy(buffer, very\_long\_string); // Buffer overflow  free(buffer);  strcpy(buffer, "test"); // Use after free |

| **Compliant Code** |
| --- |
| Use safe memory practices with bounds checking and proper lifecycle management. |
| const size\_t BUFFER\_SIZE = 10;  char\* buffer = (char\*)malloc(BUFFER\_SIZE);  if (buffer && strlen(input) < BUFFER\_SIZE) {  strncpy(buffer, input, BUFFER\_SIZE - 1);  buffer[BUFFER\_SIZE - 1] = '\0';  }  free(buffer);  buffer = nullptr; // Prevent use after free |

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

| **Principles(s):** Practice Defense in Depth: Multiple safeguards in memory handling, like bounds checking, proper allocation, and nullifying freed pointers, ensure that if one layer fails (e.g., allocation check), others still prevent exploitation. This layered approach reduces the likelihood of buffer overflows, use-after-free attacks, and memory leaks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| critical | medium | medium | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | 17.0.0 | Runtime analysis | Detects buffer overflows, use-after-free, and other memory access violations during testing. |
| Valgrind / Memcheck | 3.21.0 | Memcheck | Identifies memory leaks, invalid memory reads/writes, and improper deallocation during program execution. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Proper use of assertions helps catch programming errors during development and provides runtime checks for critical security conditions. Assertions should validate preconditions, postconditions, and invariants to ensure program correctness and prevent security vulnerabilities from logic errors. |

| **Noncompliant Code** |
| --- |
| Missing assertions on critical security conditions can allow vulnerabilities to go undetected. |
| void accessArray(int\* arr, size\_t index, size\_t size) {  // No validation of parameters  arr[index] = 42; // Potential buffer overflow  } |

| **Compliant Code** |
| --- |
| Use assertions to validate critical conditions and catch errors early. |
| void accessArray(int\* arr, size\_t index, size\_t size) {  assert(arr != nullptr);  assert(index < size);  assert(size > 0);  arr[index] = 42;  } |

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

| **Principles(s):** Use Effective QA techniques: Assertions are a proactive quality assurance measure that help detect programming errors, enforce critical conditions, and prevent logic-based security flaws. By validating assumptions during development, assertions improve code correctness and reduce the risk of vulnerabilities making it all the way to production. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | low | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | cppcoreguidelines-\* | Flags functions that lack validation checks or assertions for critical params and logic conditions. |
| Cppcheck | 2.13.0 | --enable=warning | Detects missing boundary checks, unchecked params, and potential logic flaws in functions. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling prevents information disclosure, ensures resource cleanup, and maintains program stability under error conditions. Unhandled exceptions can crash programs or leak sensitive information, while improper exception handling can create security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Unhandled exceptions can crash the program or leak sensitive information. |
| void processFile(const string& filename) {  ifstream file(filename); // May throw exception  string sensitive\_data = readCredentials();  // Exception could expose sensitive\_data in stack trace  processData(file);  } |

| **Compliant Code** |
| --- |
| Handle exceptions appropriately and ensure proper resource cleanup. |
| void processFile(const string& filename) {  try {  ifstream file(filename);  if (!file.is\_open()) {  throw runtime\_error("File access denied");  }  processData(file);  } catch (const exception& e) {  // Log error without exposing sensitive details  cerr << "File processing failed" << endl;  // Cleanup resources  }  } |

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

| **Principles(s):** Architect and Design for Security Policies: Exception handling should be planned as part of system’s architecture to ensure that errors do not expose sensitive data, cause instability, or bypass security controls. Well-designed exception policies maintain confidentiality, integrity, and availability even during failures. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | medium | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.2 | C++ Error handling Rules | Detects missing or unsafe exception handling, including cases where sensitive data may be exposed. |
| Cppcheck | 2.13.0 | --enable=warning | Flags missing try/catch blocks, potential resource leaks, and unsafe cleanup patterns for error handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-008-CPP | Comprehensive input validation is the first line of defense against many security attacks including injection attacks, buffer overflows, and malformed data attacks. All external input must be validated for type, length, format, and range before processing to ensure system security and stability. |

| **Noncompliant Code** |
| --- |
| Processing user input without validation creates multiple attack vectors. |
| void processUserData() {  string email = getUserInput();  int age = stoi(getUserInput()); // No validation  executeCommand(email); // Potential command injection  } |

| **Compliant Code** |
| --- |
| Validate all input parameters before processing using whitelisting approaches. |
| bool isValidEmail(const string& email) {  regex email\_pattern(R"([a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,})");  return regex\_match(email, email\_pattern) && email.length() <= 254;  }  void processUserData() {  string email = getUserInput();  string age\_str = getUserInput();  if (!isValidEmail(email)) {  throw invalid\_argument("Invalid email format"); }  int age = stoi(age\_str); if (age < 0 || age > 150) {  throw out\_of\_range("Invalid age range"); }  } |

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

| **Principles(s):** Validate Input Data: Every Piece of external input must be verified for type, length, format, and range before its use. Comprehensive validation prevents malicious or malformed data from triggering things like injection attacks, buffer overflows, or logic errors, ensuring secuurity and stability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| critical | high | low | critical | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | cppcoreguidelines-\* | Identifies functions that accept external input without validation and flags unsafe parameter handling. |
| OWASP ZAP | 2.14.0 | Active Scan | Performs automated injection and fuzzing tests to detect vulnerabilities caused by improper input validation. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Cryptographic Security | STD-009-CPP | Proper implementation of cryptographic functions is essential for protecting sensitive data and maintaining system security. Using weak algorithms, improper key management, or incorrect implementation of cryptographic functions can lead to data breaches and compromise system security. |

| **Noncompliant Code** |
| --- |
| Using weak or deprecated cryptographic algorithms provides insufficient security. |
| // Using weak MD5 hash for password storage  string hashPassword(const string& password) {  return md5(password); // MD5 is cryptographically broken  }    char key[] = "1234567890123456"; // Hardcoded key |

| **Compliant Code** |
| --- |
| Use strong cryptographic algorithms with proper key management and secure implementation. |
| #include <openssl/evp.h>  #include <openssl/rand.h>    string hashPassword(const string& password) {  unsigned char salt[16];  RAND\_bytes(salt, sizeof(salt));    // Use bcrypt or PBKDF2 with proper iterations  return pbkdf2\_sha256(password, salt, 10000);  }    // Generate cryptographically secure random keys  void generateKey(unsigned char\* key, size\_t length) {  if (RAND\_bytes(key, length) != 1) {  throw runtime\_error("Key generation failed");  }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard: Following established cryptographic best practices ensures strong algorithms, proper key management, and secure implementations. Avoiding deprecated algorithms and insecure storage methods protects sensitive data and preserves system confidentiality and integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| critical | medium | medium | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.2 | Cryptography Security Rules | Flags weak alorithms, hardcoded keys, and insecure cryptographic practices, recommends secure alternatives. |
| OpenSCAP | 1.3.8 | custom | Audits cryptographic configs and validates compliance with security standards like NIST. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-010-CPP | Proper resource management prevents resource leaks, denial of service attacks, and system instability. Resources including file handles, network connections, and memory must be properly acquired, used within limits, and released to maintain system security and performance. |

| **Noncompliant Code** |
| --- |
| Improper resource management can lead to resource exhaustion and system compromise. |
| void processFiles() {  while (hasMoreFiles()) {  FILE\* file = fopen(getNextFile(), "r");  // Process file but never close it  processFileData(file);  // Resource leak - file handles accumulate  }  } |

| **Compliant Code** |
| --- |
| Use RAII principles and proper resource management to ensure cleanup. |
| class FileHandler {  FILE\* file;  public:  FileHandler(const char\* filename) : file(fopen(filename, "r")) {  if (!file) throw runtime\_error("Failed to open file");  }  ~FileHandler() { if (file) fclose(file); }  FILE\* get() const { return file; }  };    void processFiles() {  try {  while (hasMoreFiles()) {  FileHandler handler(getNextFile());  processFileData(handler.get());  // Automatic cleanup via destructor  }  } catch (const exception& e) {  cerr << "Error processing files: " << e.what() << endl;  }  } |

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

| **Principles(s):** Practice Defense in Depth: Proper resource management ensures that multiple safeguards are in place to prevent leaks, exhaustion, and instability. By using patterns like RAII and enforcing cleanup at all stages, the system maintains performance and resilience against attacks exploiting resource limits. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | low | high | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | Clang-analyzer-cplusplus.NewDeleteLeaks | Detects unreleased resources, missing destructors, improper cleanup in RAII and manual management code. |
| Cppcheck | 2.13.0 | --enable=warning | Finds potential leaks of file handles and memory by analyzing control flow and resource usage. |

### 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 enforces and monitors compliance with the standards defined in this policy by embedding security controls into every phase of the DevSecOps cycle. The infinity loop represents the continuing, iterative nature of this process, ensuring that security is not a 1x step but an ongoing practice. The pre-production phases (Assess/Plan, Design, Build, Verify/Test),automated tools like Clang-Tidy, dependency scans, and vulnerability scanners will run in the CI/CD pipeline to detect violations of secure coding standards before code is merged and/or deployed. So builds will auto-fail if noncompliant code is detected.

The production phases (Transition, Monitor, Respond, Maintain), automation focuses on continuous monitoring, logging, and enforcement of the Triple-A framework. This includes SIEM integration, intrusion detection, and auto remediation scripts to change back noncompliant configs or block malicious activity. The existing DevOps process should be modified by adding security scanning stages to the pipeline, integrating pre-commit hooks for secure coding checks, and managing compliance as code in version-controlled repos. This ensures that security will remain ongoing and automated rather than one time.

### 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 | medium | low | high | 4 |
| STD-002-CPP | high | high | low | high | 5 |
| STD-003-CPP | high | high | low | high | 5 |
| STD-004-CPP | critical | high | low | critical | 5 |
| STD-005-CPP | critical | med | med | high | 5 |
| STD-006-CPP | high | med | low | high | 4 |
| STD-007-CPP | high | med | med | high | 4 |
| STD-008-CPP | critical | high | low | critical | 5 |
| STD-009-CPP | critical | med | med | high | 4 |
| STD-010-CPP | high | med | low | high | 4 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects data stored on physical or virtual media, like databases, file systems, backups, or cloud storage. Keys must be securely stored in a hardware security module (HSM) or equivalent key management service. This policy applies to all storage systems containing sensitive information, including production servers, development environments, employee endpoints, and removable media among others. |
| Encryption in flight | Encryption in flight protects data as it moves between things, preventing interception/tampering during transmission. Require secure transport protocols such as TLS 1.3 or higher for all data exchanges over networks, including API calls, web traffic, email, and internal service communication can help. Encryption in flight ensures data confidentiality and integrity against man-in-the-middle attacks, packet sniffing, and replay attacks. |
| Encryption in use | Encryption in use protects data while it is actively being processed in memory, reducing the risk of exposure through memory scraping, side-channel attacks, or unauthorized debugging. Apply memory encryption where supported by hardware, and minimize plaintext exposure duration in application workflows helps protect it. Encryption in use mitigates risks from insider threats, compromised hosts, or malware with memory access privileges. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user’s or system’s identity before granting access to resources. This policy applies to all access points, like application logins, VPN connections, administrative consoles, and API endpoints. |
| Authorization | Authorization allows what actions or resources an authenticated user is permitted to access based on their role or permissions. Proper authorization controls reduce the risk of insider threats, privilege escalation, and accidental misuse, among other issues. |
| Accounting | Accounting (or auditing) tracks and logs user actions to provide a record of activity for security, compliance, and forensic analysis. Accounting enables proactive detection of suspicious activity and ensures accountability for all user action that can be tracked. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 8/12/2025 | Principles and standards built out | Amanda Stone | Amanda Stone |
| [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 |