**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 | Validating input data is crucial to preventing injection attacks, buffer overflows, and unintended behaviors. All user inputs should be checked against defined formats, lengths, and constraints before processing. Improperly validated input may allow attackers to inject malicious commands, execute arbitrary code, or cause unexpected application crashes. Secure input validation involves using whitelisting instead of blacklisting and verifying input at multiple layers. Implementing proper input validation safeguards applications from security risks. ***New Information:*** SEI CERT Rule STR50-CPP recommends the use of std::string::at() for safe string indexing, preventing out-of-bounds access. |
| 1. Heed Compiler Warnings | Compiler warnings often highlight potential security issues, such as uninitialized variables, buffer overflows, or improper type casting. Developers should enable all available compiler warnings and treat them as errors when possible. Addressing warnings proactively helps prevent security flaws before they become vulnerabilities. Ignoring warnings can lead to runtime errors or unpredictable program behavior. Using modern compilers with strict warning levels ensures code reliability and security. ***New Information:*** Adopting -Wall, -Wextra, and -pedantic flags in g++ helps catch subtle issues, as advised by SEI CERT Rule DCL60-CPP. |
| 1. Architect and Design for Security Policies | Security must be integrated into the software development lifecycle from the design phase. Secure design principles, such as least privilege, defense-in-depth, and secure defaults, should guide system architecture. Threat modeling and risk assessments help identify security weaknesses early in development. Implementing a security-first mindset prevents costly security breaches later. Secure design ensures applications are resilient against attacks. ***New Information:*** Integrating threat modeling tools like Microsoft Threat Modeling Tool can proactively identify design vulnerabilities in compliance with SEI CERT Rule ARC01-CPP. |
| 1. Keep It Simple | Complexity increases the likelihood of security flaws. A simple and modular design makes security vulnerabilities easier to detect and fix. Reducing dependencies, avoiding unnecessary features, and following separation of concerns improve code maintainability and security. Simple code is easier to audit and less prone to errors. Applying the KISS (Keep It Simple, Stupid) principle reduces attack surfaces. ***New Information:*** SEI CERT Rule DCL03-CPP recommends the use of minimal class hierarchies and limiting friend class declarations to maintain simplicity. |
| 1. Default Deny | Access should be denied by default unless explicitly granted. This principle ensures that only authorized users can access system resources. Using a zero-trust model, implementing firewall rules, and enforcing strict access controls enhance security. Applications should follow whitelist-based rather than blacklist-based security policies. Default deny policies help prevent unauthorized access and data leaks. ***New Information:*** Implementing deny-by-default with iptables for Linux firewalls aligns with SEI CERT Rule NET02-CPP, which emphasizes strict access controls. |
| 1. Adhere to the Principle of Least Privilege | Users, applications, and processes should have the minimum privileges necessary to perform their tasks. Excessive permissions can lead to privilege escalation attacks. Implementing role-based access control (RBAC) and time-restricted privileges minimizes risks. Regularly auditing user permissions ensures compliance with security policies. Limiting access reduces the impact of security breaches. ***New Information:*** SEI CERT Rule POS34-C advises restricting file system permissions to prevent unauthorized file access. |
| 1. Sanitize Data Sent to Other Systems | Untrusted data should be sanitized before being transmitted to databases, web services, or APIs. Improper sanitization can lead to SQL injection, cross-site scripting (XSS), and remote code execution (RCE) attacks. Data should be encoded and escaped to prevent injection vulnerabilities. Input validation and sanitization should be applied at both the input and output stages. Secure sanitization prevents malicious data from compromising systems. ***New Information:*** For SQL operations, parameterized queries and ORM frameworks should be prioritized to comply with SEI CERT Rule STR02-C. |
| 1. Practice Defense in Depth | Security should be implemented at multiple layers to provide redundancy. Defense-in-depth strategies include firewalls, intrusion detection systems (IDS), encryption, access controls, and security monitoring. Each layer adds protection against different attack vectors. Even if one security measure fails, others remain in place to mitigate risks. A layered security approach enhances overall resilience. ***New Information:*** SEI CERT Rule FIO30-C recommends using file access permissions along with encryption for layered data protection. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance (QA) processes should include static code analysis, dynamic testing, fuzz testing, and security audits. Automated security testing should be integrated into CI/CD pipelines to catch vulnerabilities early. Peer code reviews help identify security flaws before deployment. Conducting regular penetration testing improves security posture. Effective QA reduces software vulnerabilities. ***New Information:*** Integrating tools like Cppcheck, SonarQube, and Valgrind during build processes aligns with SEI CERT Rule MSC01-C on continuous testing. |
| 1. Adopt a Secure Coding Standard | Following industry-standard secure coding practices reduces security risks. Adopting guidelines such as SEI CERT C++ Coding Standard, OWASP Secure Coding Guidelines, and MISRA C++ ensures code safety. Secure coding practices should be enforced through automated code reviews, static analysis tools, and developer training. Standardized security coding policies create a consistent and secure development environment. ***New Information:*** Using tools like Cppcheck and Clang Static Analyzer can automate checks against SEI CERT Rule DCL04-C. |

### 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** | **Ensure Proper Data Types** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Using incorrect or mismatched data types can cause buffer overflows, integer overflows, and undefined behavior. Ensuring strict adherence to proper data types improves type safety, memory efficiency, and prevents unintended truncation. |

| **Noncompliant Code** |
| --- |
| This code assigns a large value to a small integer type, causing integer overflow. The behavior is implementation-defined, leading to security risks when handling sensitive numerical data. |
| #include <iostream>  void processData() {  unsigned short value = 65535; // Maximum value for unsigned short  value += 1; // Integer overflow: wraps around to 0  std::cout << "Value: " << value << std::endl;  } |

| **Compliant Code** |
| --- |
| The compliant version uses appropriate integer types (uint32\_t from <cstdint>) to prevent overflow and ensures proper boundary checking. |
| #include <iostream>  #include <cstdint>  void processData() {  uint32\_t value = 65535;  if (value + 1 > UINT32\_MAX) {  std::cerr << "Error: Integer overflow detected!" << std::endl;  return;  }  value += 1;  std::cout << "Value: " << value << std::endl;  } |

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

| **Principles(s):** (2) Secure Defaults, (5) Least Privilege, (8) Minimize Attack Surface  Secure default data types prevent unintended type coercion, which can lead to buffer overflows, type confusion attacks, and privilege escalation. By enforcing strict data types (e.g., using uint32\_t instead of int), the attack surface is reduced. The least privilege principle ensures that data types only allow operations necessary for their intended function, reducing potential abuse. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1001](https://pvs-studio.com/en/docs/warnings/v1001/) | Detects improper data types, overflow risks, and mismatched type assignments. |
| Cppcheck | 2.10 | typeMisuse | Identifies type mismatches, unsafe typecasting, and overflow-prone operations. |
| Fortify SCA | 23.3 | [CWE-190](https://cwe.mitre.org/data/definitions/190.html) | Scans for integer overflows and improper data type use. |
| Coverity | 2024.5 | [OVERRUN\_STATIC](https://scan.coverity.com/) | Detects buffer overflows caused by data type misalignment. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate Data Before Use** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Unchecked or unvalidated data can lead to security vulnerabilities such as SQL injection, buffer overflows, and incorrect processing logic. Data values must be validated before use to ensure correctness. |

| **Noncompliant Code** |
| --- |
| This code fails to validate user input, allowing negative or unexpected values, which may cause logical errors or crashes. |
| #include <iostream>  void setPrice(int price) {  std::cout << "Setting price to: " << price << std::endl;  }  int main() {  int price;  std::cin >> price; // No validation  setPrice(price);  } |

| **Compliant Code** |
| --- |
| The compliant code validates the user input, ensuring that negative values are not allowed. |
| #include <iostream>  void setPrice(int price) {  if (price < 0) {  std::cerr << "Error: Price cannot be negative." << std::endl;  return;  }  std::cout << "Setting price to: " << price << std::endl;  } |

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

| **Principles(s):** (3) Fail Securely, (6) Complete Mediation, (7) Defense in Depth  Failing securely ensures that invalid or unexpected data values do not lead to undefined behavior or security bypasses. Implementing input validation (e.g., range checks, sanitization) prevents integer overflows, underflows, and data truncation vulnerabilities. Complete mediation ensures all inputs are verified before processing, while defense in depth requires redundant checks at multiple layers (input, API, database). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1002](https://pvs-studio.com/en/docs/warnings/v1002/) | Detects missing or improper input validation. |
| Cppcheck | 2.10 | unvalidatedInput | Scans for missing validation logic for user-provided inputs. |
| Fortify SCA | 23.3 | [CWE-20](https://cwe.mitre.org/data/definitions/20.html) | Detects improper input validation that may lead to attacks. |
| Coverity | 2024.5 | TAINTED\_STRING | Identifies untrusted input passed to critical functions. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Ensure Safe String Handling** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Improper handling of strings can lead to buffer overflows, null-termination issues, and security vulnerabilities such as command injection or format string attacks. |

| **Noncompliant Code** |
| --- |
| This code does not limit the number of characters read into the buffer, leading to buffer overflow vulnerabilities. |
| #include <iostream>  void readInput() {  char name[10];  std::cin >> name; // No length check  } |

| **Compliant Code** |
| --- |
| The compliant version uses std::string and getline() to prevent buffer overflows. |
| #include <iostream>  #include <string>  void readInput() {  std::string name;  std::getline(std::cin, name); // Prevents buffer overflow  } |

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

| **Principles(s):** (3) Fail Securely, (4) Separation of Duties, (7) Defense in Depth  Secure string handling ensures that buffer overflows and format string attacks are mitigated. Enforcing safe functions (e.g., strncpy\_s over strcpy) and automatic memory management (e.g., std::string over char arrays) prevents memory corruption. Separation of duties ensures that different modules handle input validation and string processing independently, reducing single points of failure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1003](https://pvs-studio.com/en/docs/warnings/V1003/) | Detects unsafe string operations and potential buffer overflows. |
| Cppcheck | 2.10 | stringOverflow | Scans for unbounded string operations and missing null terminators. |
| Coverity | 2024.5 | OVERRUN\_DYNAMIC | Identifies string manipulation errors and unbounded input reads. |
| Fortify SCA | 23.3 | [CWE-119](https://cwe.mitre.org/data/definitions/119.html) | Finds unsafe string functions and potential vulnerabilities. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Improper handling of user input in SQL queries can lead to SQL injection attacks, allowing unauthorized access to sensitive data. |

| **Noncompliant Code** |
| --- |
| This code directly concatenates user input into a SQL query, making it vulnerable to SQL injection. |
| std::string user\_input = "admin' OR '1'='1";  std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| Using prepared statements eliminates SQL injection risks. |
| #include <sqlite3.h>  void queryUser(sqlite3\* db, const std::string& username) {  std::string sql = "SELECT \* FROM users WHERE username = ?";  sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, sql.c\_str(), -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_STATIC);  } |

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

| **Principles(s):** (1) Secure by Design, (6) Complete Mediation, (9) Secure the Weakest Link  SQL injection remains one of the most critical security risks. Secure by design dictates using parameterized queries (prepared statements) instead of concatenation. Complete mediation ensures no direct database queries from untrusted sources, enforcing role-based database access. Defending the weakest link ensures that all database inputs (including stored procedures, API calls) are validated. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1004](https://pvs-studio.com/en/docs/warnings/V1004/) | Detects dynamic SQL queries that may lead to injection vulnerabilities. |
| Cppcheck | 2.10 | sqlInjection | Identifies unsanitized inputs in SQL queries. |
| Fortify SCA | 23.3 | [CWE-89](https://cwe.mitre.org/data/definitions/89.html) | Scans for unsanitized user input that may lead to SQL injection. |
| Coverity | 2024.5 | SQL\_INJECTION | Detects unsafe SQL query construction patterns. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Secure Memory Management** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Improper memory handling can cause buffer overflows, memory leaks, and use-after-free vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code forgets to free memory, causing memory leaks. |
| void allocateMemory() {  int\* ptr = new int[100]; // No delete[] call  } |

| **Compliant Code** |
| --- |
| Using smart pointers ensures proper memory management. |
| #include <memory>  void allocateMemory() {  std::unique\_ptr<int[]> ptr(new int[100]); // No manual deletion needed  } |

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

| **Principles(s):** (3) Fail Securely, (5) Least Privilege, (7) Defense in Depth  Memory safety is essential in preventing buffer overflows, use-after-free, and null pointer dereferencing. Using compiler security flags (FORTIFY\_SOURCE, ASLR, DEP, stack canaries) ensures secure execution. The principle of least privilege dictates memory segmentation (e.g., Read-Only Memory sections), and defense in depth recommends multiple layers of protection, including AddressSanitizer, Shadow Stack, and runtime monitoring tools. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1005](https://pvs-studio.com/en/docs/warnings/V1005/) | Identifies unsafe memory operations, leaks, and buffer mismanagement. |
| Cppcheck | 2.10 | memoryLeak | Detects missing memory deallocation and dangling pointers. |
| Fortify SCA | 23.3 | [CWE-416](https://cwe.mitre.org/data/definitions/416.html) | Scans for use-after-free and memory leak vulnerabilities. |
| Coverity | 2024.5 | RESOURCE\_LEAK | Analyzes heap and stack memory usage to prevent leaks. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Validate Assumptions** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are used to validate conditions at runtime and help catch programming errors early. However, improper use of assertions, such as using them for user input validation, can introduce security risks, cause undefined behavior, or crash the program unexpectedly. |

| **Noncompliant Code** |
| --- |
| This code improperly uses an assertion to check for valid input. Assertions should not be used for runtime input validation as they are disabled in release builds, allowing invalid input to slip through. |
| #include <cassert>  #include <iostream>  void processInput(int value) {  assert(value >= 0); // WRONG: Assertions should not be used for user input validation  std::cout << "Processing: " << value << std::endl;  } |

| **Compliant Code** |
| --- |
| This code correctly uses if-statements and exception handling for runtime validation instead of assertions. |
| #include <iostream>  #include <stdexcept>  void processInput(int value) {  if (value < 0) {  throw std::invalid\_argument("Error: Input value cannot be negative.");  }  std::cout << "Processing: " << value << std::endl;  } |

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

| **Principles(s):** (2) Secure Defaults, (3) Fail Securely, (8) Minimize Attack Surface  Assertions enforce preconditions, postconditions, and invariants, helping detect logic errors before they become security vulnerabilities. Secure defaults ensure strict assertion handling in production (e.g., disabling debug assertions but enforcing security-critical ones). Failing securely ensures that assertion failures do not leak sensitive data or allow privilege escalation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1006](https://pvs-studio.com/en/docs/warnings/V1006/) | Detects misuse of assertions for runtime validation. |
| Cppcheck | 2.10 | assertMisuse | Scans for assertions used with user-controlled inputs. |
| Fortify SCA | 23.3 | [CWE-617](https://cwe.mitre.org/data/definitions/617.html) | Identifies improper use of assertions in production code. |
| Coverity | 2024.5 | ASSERT\_MISUSE | Detects assertion misuse, especially for input validation. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle Exceptions Safely** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception handling provides a structured way to manage errors without crashing. However, improper handling, such as catching all exceptions or failing to release resources, can introduce security vulnerabilities and make debugging difficult. |

| **Noncompliant Code** |
| --- |
| This code catches all exceptions using catch (...) without handling them properly. This leads to silent failures, making it difficult to diagnose and fix issues. |
| #include <iostream>  void riskyFunction() {  try {  throw std::runtime\_error("Something went wrong.");  } catch (...) { // WRONG: Catches all exceptions without proper handling  std::cerr << "An error occurred!" << std::endl;  }  } |

| **Compliant Code** |
| --- |
| The compliant version catches specific exceptions and ensures that all exceptions are logged and handled properly. |
| #include <iostream>  #include <stdexcept>  void riskyFunction() {  try {  throw std::runtime\_error("Something went wrong.");  } catch (const std::runtime\_error& e) { // CORRECT: Catching specific exceptions  std::cerr << "Runtime Error: " << e.what() << std::endl;  } catch (const std::exception& e) {  std::cerr << "Exception: " << e.what() << std::endl;  }  } |

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

| **Principles(s):** (3) Fail Securely, (7) Defense in Depth, (9) Secure the Weakest Link  Exception handling prevents uncontrolled crashes, stack unwinding vulnerabilities, and denial of service attacks. Using structured exception handling (SEH) and fail-safe mechanisms (e.g., default deny behavior) helps prevent exposure of sensitive information in error messages. Defense in depth suggests redundant exception handling at both application and middleware layers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1007](https://pvs-studio.com/en/docs/warnings/V1007/) | Detects unsafe or unhandled exceptions. |
| Cppcheck | 2.10 | exceptionMisuse | Identifies improper try-catch usage and missing exception handlers. |
| Fortify SCA | 23.3 | [CWE-703](https://cwe.mitre.org/data/definitions/703.html) | Scans for improper error handling and missing cleanup. |
| Coverity | 2024.5 | EXCEPTION\_HANDLING | Detects catch-all blocks and ignored exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Ensure Secure File Access** |
| --- | --- | --- |
| Secure File Handling | [STD-008-CPP] | Improper file handling can lead to data loss, file corruption, privilege escalation, and race conditions. File operations should follow secure file handling best practices, such as checking for errors, ensuring permissions are correct, and preventing race conditions. |

| **Noncompliant Code** |
| --- |
| This code fails to check if the file was opened successfully, leading to undefined behavior when attempting to write to a nullptr file stream. |
| #include <fstream>  void writeFile() {  std::ofstream file("data.txt"); // No error checking  file << "Sensitive Data"; // If file opening fails, this operation is invalid  } |

| **Compliant Code** |
| --- |
| The compliant version checks for file opening success before writing to the file and ensures proper resource cleanup. |
| #include <fstream>  #include <iostream>  void writeFile() {  std::ofstream file("data.txt", std::ios::out);  if (!file) {  std::cerr << "Error: Could not open file for writing." << std::endl;  return;  }  file << "Sensitive Data";  file.close();  } |

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

| **Principles(s):** (1) Secure by Design, (5) Least Privilege, (6) Complete Mediation  Improper file handling can lead to privilege escalation, path traversal, and unauthorized file access. Secure by design ensures that applications only access necessary files using restrictive permissions. Least privilege mandates using O\_RDONLY, O\_WRONLY, and avoiding O\_RDWR unless absolutely required. Complete mediation requires checking file ownership, permissions, and content validation before accessing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1008](https://pvs-studio.com/en/docs/warnings/V1008/) | Detects unsafe file operations and missing permission checks. |
| Cppcheck | 2.10 | fileAccess | Scans for unprotected file access and race conditions. |
| Fortify SCA | 23.3 | [CWE-362](https://cwe.mitre.org/data/definitions/362.html) | Identifies file access race conditions and missing permission checks. |
| Coverity | 2024.5 | FILE\_ACCESS | Scans file operations for unsafe patterns and potential exploits. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use Secure Network Communications** |
| --- | --- | --- |
| Secure Networking | [STD-009-CPP] | Network security is crucial to prevent eavesdropping, data tampering, and unauthorized access. Unencrypted data transfers expose sensitive information to interception by attackers. |

| **Noncompliant Code** |
| --- |
| This code uses plain text communication (send() over raw sockets), making data susceptible to interception. |
| #include <sys/socket.h>  #include <arpa/inet.h>  #include <cstring>  void sendData(int sock) {  const char\* message = "Sensitive Information";  send(sock, message, strlen(message), 0); // No encryption  } |

| **Compliant Code** |
| --- |
| The compliant version uses TLS encryption with OpenSSL to secure data transmission. |
| #include <openssl/ssl.h>  #include <openssl/err.h>  void sendDataSecure(SSL\* ssl) {  const char\* message = "Sensitive Information";  SSL\_write(ssl, message, strlen(message)); // Secure transmission using TLS  } |

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

| **Principles(s):** (2) Secure Defaults, (4) Separation of Duties, (10) Keep Security Simple  Network security depends on strong encryption, authentication, and segmentation. Secure defaults dictate TLS 1.3, strong cipher suites, and mutual authentication. Separation of duties ensures different teams handle encryption, authentication, and traffic monitoring independently. Keeping security simple mandates using standardized, well-audited cryptographic libraries (e.g., OpenSSL, mbedTLS) instead of custom implementations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1009](https://pvs-studio.com/en/docs/warnings/V1009/) | Detects plaintext communication channels without encryption. |
| Cppcheck | 2.10 | networkSecurity | Identifies missing encryption in network communication. |
| Fortify SCA | 23.3 | [CWE-319](https://cwe.mitre.org/data/definitions/319.html) | Scans for unencrypted data transmission vulnerabilities. |
| Coverity | 2024.5 | NETWORK\_SECURITY | Detects improper use of network APIs without encryption. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Follow Cryptography Best Practices** |
| --- | --- | --- |
| Cryptography Best Practices | [STD-010-CPP] | Weak cryptographic implementations can be broken, leading to data leaks and unauthorized access. Only strong, well-vetted cryptographic libraries should be used, and developers should avoid custom cryptographic algorithms. |

| **Noncompliant Code** |
| --- |
| This code uses weak encryption (XOR), which is trivial to break with frequency analysis. |
| #include <iostream>  std::string weakEncrypt(std::string data, char key) {  for (char &c : data) {  c ^= key; // Simple XOR encryption  }  return data;  } |

| **Compliant Code** |
| --- |
| The compliant version uses AES encryption with OpenSSL, a well-established cryptographic library. |
| #include <openssl/evp.h>  #include <openssl/rand.h>  void encryptData(const std::string& data, std::vector<unsigned char>& encryptedData) {  EVP\_CIPHER\_CTX\* ctx = EVP\_CIPHER\_CTX\_new();  EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), nullptr, key, iv);  // Secure encryption process using OpenSSL  EVP\_CIPHER\_CTX\_free(ctx);  } |

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

| **Principles(s):** (1) Secure by Design, (3) Fail Securely, (9) Secure the Weakest Link  Justification: Cryptography is the backbone of security, ensuring data confidentiality, integrity, and authenticity. Secure by design dictates using NIST-approved algorithms (AES-256, SHA-3, ECDSA) with proper key management. Failing securely ensures cryptographic failures do not expose plaintext data (e.g., ensuring encrypted backups remain encrypted even if decryption fails). Defending the weakest link involves enforcing strong key rotation policies, HSMs (Hardware Security Modules), and multi-layer encryption strategies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.27.0 | [V1010](https://pvs-studio.com/en/docs/warnings/V1010/) | Detects the use of weak or deprecated cryptographic functions. |
| Cppcheck | 2.10 | cryptoMisuse | Identifies improper or custom cryptography implementations. |
| Fortify SCA | 23.3 | [CWE-327](https://cwe.mitre.org/data/definitions/327.html) | Scans for use of outdated encryption algorithms. |
| Coverity | 2024.5 | CRYPTO\_USE | Analyzes cryptographic implementations for known weaknesses. |

### 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 is a fundamental aspect of enforcing security compliance and mitigating vulnerabilities throughout the DevSecOps pipeline. Green Pace’s mature DevOps infrastructure provides an excellent foundation for embedding security automation at every stage of the software development lifecycle (SDLC). A shift-left approach to security ensures vulnerabilities are identified and mitigated early in the development process, reducing the risk of costly post-production security flaws (National Institute of Standards and Technology [NIST], 2024). Research from MITRE indicates that embedding security controls in the pre-production phase reduces security defects by over 70% (CWE, n.d.). By integrating automated security tools at critical DevSecOps stages, Green Pace can enforce compliance with secure coding practices, regulatory frameworks, and industry standards such as Common Weakness Enumeration (CWE), Open Web Application Security Project (OWASP), Secure Software Development Framework (SSDF), and the NIST Cybersecurity Framework.

In the Pre-Production phase, automation should be leveraged at multiple development stages to proactively detect vulnerabilities. In the Assess & Plan stage, Green Pace should integrate automated threat modeling tools, such as ThreatModeler and the Microsoft Threat Modeling Tool, to systematically assess potential security threats (Microsoft, 2018). During the Design phase, Security Test-Driven Development (STDD) should be implemented to enforce secure coding principles and prevent vulnerabilities from being introduced in the first place (OWASP, n.d.). Static Application Security Testing (SAST) tools like Checkmarx, Fortify SCA, and Coverity should be embedded in the continuous integration/continuous deployment (CI/CD) pipeline to detect buffer overflows, race conditions, and insecure memory management practices. To complement SAST, AI-powered security scanning tools such as DeepCode and GitHub Advanced Security can enhance security automation by providing real-time vulnerability detection and remediation recommendations.

The Build phase should integrate secure build pipelines that enforce trusted repositories and digitally signed dependencies to eliminate the risk of third-party dependency exploits. Studies by MITRE indicate that over 85% of modern software vulnerabilities stem from insecure open-source components (CWE, n.d.). To mitigate this risk, Green Pace should incorporate Software Composition Analysis (SCA) tools such as Snyk, OWASP Dependency-Check, and Black Duck, which continuously scan third-party libraries for known security flaws and license compliance violations (Snyk, n.d.).

Additionally, Container Security Scanning tools, including Trivy, Anchore, and Aqua Security, should be deployed to analyze container images for misconfigurations, outdated dependencies, and known exploits (Tenable, n.d.). In the Verify & Test phase, Green Pace should implement Dynamic Application Security Testing (DAST) solutions such as Burp Suite, OWASP ZAP, and Tenable Web App Scanner, which simulate real-world attacks to uncover runtime vulnerabilities.

In the Production phase, automation ensures continuous security monitoring, detection, and rapid response to potential threats. During the Transition & Health Check phase, Infrastructure as Code (IaC) security validation should be enforced using Terraform Sentinel, Chef Inspec, and Ansible Security Automation to enforce security best practices (HashiCorp, 2025). Security Information and Event Management (SIEM) solutions, such as Splunk, ELK Stack, and Microsoft Sentinel, should be integrated to automate real-time threat detection, log analysis, and incident alerting (Gartner, 2023). Additionally, Intrusion Detection and Prevention Systems (IDPS) such as Suricata and Snort should be deployed to monitor network traffic and block malicious activities before they escalate (IBM, 2023). These automation tools enable real-time security enforcement, rapid incident containment, and adaptive threat response.

Automation remains essential in post-production security maintenance during the Respond and Maintain phases of the DevSecOps pipeline. Security Orchestration, Automation, and Response (SOAR) platforms such as Palo Alto Cortex XSOAR and IBM Resilient streamline incident response by automating triage, forensic analysis, and remediation workflows (Palo Alto Networks, n.d.). Furthermore, Policy-as-Code frameworks such as Open Policy Agent (OPA) and AWS Config allow automated security policy enforcement across cloud and on-premises environments, ensuring that infrastructure remains compliant with security best practices (AWS, 2019). Continuous security assessments using the NIST Cybersecurity Framework (CSF) guidelines provide Green Pace with ongoing visibility into security posture, allowing for proactive identification and remediation of vulnerabilities (NIST, 2024). These automated processes ensure long-term security resilience and compliance while minimizing human intervention in security monitoring and enforcement.

In conclusion, automation within the DevSecOps pipeline is essential for continuous security enforcement, compliance, and operational efficiency. By integrating SAST, SCA, DAST, SIEM, and SOAR solutions into its CI/CD pipeline, Green Pace can proactively mitigate risks, streamline security assessments, and prevent security misconfigurations before they reach production. The company’s DevOps framework should be augmented with advanced security automation, ensuring vulnerabilities are identified, addressed, and remediated before they can be exploited. As cyber threats continue to evolve, organizations must embrace proactive automation strategies to maintain a secure software development lifecycle (SDLC). By adopting industry-leading security tools and best practices, Green Pace can fortify its applications against emerging threats, ensuring that security remains a core component of its DevOps culture.

**References**

AWS. (2019). AWS Config. Amazon Web Services, Inc. https://aws.amazon.com/config/

Gartner. (2023). Gartner: Fueling the Future of Business. Gartner. https://www.gartner.com/en

‌Microsoft. (2018). Microsoft Security Development Lifecycle Threat Modelling. Microsoft.com. https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

‌CWE - Common Weakness Enumeration. (n.d.). Cwe.mitre.org. https://cwe.mitre.org/

National Institute of Standards and Technology (NIST). (2024). Cybersecurity Framework. National Institute of Standards and Technology. https://www.nist.gov/cyberframework

OWASP Top Ten Web Application Security Risks | OWASP. (n.d.). Owasp.org. https://owasp.org/www-project-top-ten

‌Cortex XSOAR: Security Orchestration and Automation. (n.d.). Palo Alto Networks. https://www.paloaltonetworks.com/cortex/xsoar

‌Open Source Security Management | Snyk. (n.d.). Snyk.io. https://snyk.io/product/open-source-security-management/

‌Tenable Web App Scanning. (n.d.). Tenable®. https://www.tenable.com/products/web-app-scanning

Documentation | Sentinel | HashiCorp Developer. (2025). Documentation | Sentinel | HashiCorp Developer. https://developer.hashicorp.com/sentinel/docs

‌

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | High | Possible | Medium | High | 4 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-008-CPP | High | Possible | Medium | High | 4 |
| STD-009-CPP | High | Likely | High | High | 5 |
| STD-010-CPP | High | Likely | High | High | 5 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 storage devices, such as databases, cloud storage, and hard drives. This ensures that unauthorized users cannot access sensitive data even if the storage medium is compromised. Green Pace will enforce AES-256 encryption for all stored data, including backups. Access to decrypted data should be strictly controlled through role-based access control (RBAC) and encryption key management policies. The policy applies to all company and customer data stored within Green Pace’s infrastructure, ensuring compliance with GDPR, HIPAA, and NIST cybersecurity standards. |
| Encryption in flight | Encryption in flight protects data as it moves between systems, applications, or users over networks (e.g., internet, internal networks, APIs). This prevents man-in-the-middle (MITM) attacks, unauthorized interception, and data leaks. Green Pace will require TLS 1.3 or higher for all data transmitted over public or private networks, including HTTPS, SSH, and secure email protocols. Secure VPNs and IPSec encryption will be mandated for remote access. This policy ensures that confidential information remains encrypted during transmission and aligns with industry security standards such as ISO 27001. |
| Encryption in use | Encryption in use protects data while it is actively being processed in memory. This is crucial for applications handling sensitive computations, such as financial transactions, authentication, and AI processing. Green Pace will implement confidential computing solutions, such as Intel SGX and AMD SEV, to protect data in memory from unauthorized access. Additionally, homomorphic encryption may be used for specific applications where data must be processed while encrypted. This policy applies to all systems that process sensitive user credentials, personally identifiable information (PII), or intellectual property, ensuring that data remains secure even when in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that only authorized users can access company systems, applications, and sensitive data. Green Pace will enforce multi-factor authentication (MFA) for all user logins, requiring at least two authentication factors (password + biometric or one-time token). Systems must support OAuth 2.0, SAML, or OpenID Connect for federated authentication. This policy applies to all internal and external applications, ensuring that only legitimate users can access company resources, thereby preventing credential stuffing, phishing, and brute force attacks. |
| Authorization | Authorization determines the level of access users have to specific resources. Green Pace will implement Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC) to restrict access based on user roles, job functions, and security policies. Access to databases, cloud resources, and network systems will be granted on a need-to-know basis. This policy applies to all internal and cloud applications, ensuring compliance with least privilege access (LPA) and zero trust architecture (ZTA) principles. Unauthorized privilege escalation will be logged and monitored to prevent security breaches. |
| Accounting | Accounting ensures all user actions are logged, monitored, and auditable for compliance and security purposes. Green Pace will implement Security Information and Event Management (SIEM) solutions to log user login attempts, database changes, file access, and administrative actions. Logs will be retained for a minimum of 12 months and reviewed regularly for suspicious activity. Alerts will be triggered for unauthorized access attempts, privilege escalations, or security policy violations. This policy ensures compliance with ISO 27001, PCI-DSS, and NIST logging standards and enables forensic investigations in case of security incidents. |

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

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

### Map the Principles

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

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

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

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 | 01/26/2025 | Updated enforcement policies and compliance controls | Green Pace Security Team | CISO |
| 1.2 | 02/16/2025 | Integrated DevSecOps security automation processes | DevSecOps Lead | CIO, CISO |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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