**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 for 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 | Always validate input data to ensure it meets the expected format and constraints before processing. Unchecked input can lead to security vulnerabilities such as buffer overflows, SQL injection, or cross-site scripting (XSS). Proper input validation protects against malicious attacks and ensures that only trusted data is processed. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential issues in the code, including unsafe operations, possible security vulnerabilities, or deprecated functions. Developers should pay close attention to these warnings, resolve them, and follow best practices for secure coding to avoid introducing flaws that could be exploited. |
| 1. Architect and Design for Security Policies | Security should be embedded into the architecture and design phase of software development. By applying security policies early on, such as secure data handling, proper access control, and secure communication channels, the system can be built to withstand potential threats from the outset, reducing the need for costly changes later in development. |
| 1. Keep It Simple | Simplicity in design and code leads to easier maintenance, better readability, and fewer chances for security flaws to emerge. Complex systems or convoluted code can introduce unexpected vulnerabilities. By following the principle of simplicity, developers ensure that the application remains more secure, understandable, and easier to audit. |
| 1. Default Deny | The "default deny" principle means that access is denied unless explicitly allowed. In practice, this implies that systems and applications should assume that all access requests are malicious until they are properly authenticated and authorized. This reduces the attack surface and ensures that only legitimate users or services can access resources. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege dictates that users, applications, and systems should only be given the minimum level of access necessary to perform their tasks. By limiting access rights, the potential damage from a compromised account or process is minimized. This principle helps reduce the risk of internal and external security breaches. |
| 1. Sanitize Data Sent to Other Systems | When transmitting data to external systems or services, it is crucial to sanitize the data to prevent injection attacks and other vulnerabilities. Data should be validated, encoded, or filtered to ensure that it does not contain harmful content or malicious code that could exploit weaknesses in other systems. |
| 1. Practice Defense in Depth | Defense in depth is a security strategy that employs multiple layers of protection. If one layer is breached, other layers continue to protect the system. This could involve using firewalls, encryption, access controls, and intrusion detection systems in conjunction to safeguard the application against various types of threats. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance (QA) is essential to identify security flaws before deployment. Effective QA includes comprehensive testing practices like static analysis, dynamic analysis, and penetration testing. These techniques help identify vulnerabilities, bugs, and weaknesses early in the development cycle, improving the overall security and reliability of the software. |
| 1. Adopt a Secure Coding Standard | A secure coding standard is a set of guidelines that provides best practices and rules for writing code that minimizes security risks. Adopting a secure coding standard ensures that developers consistently implement secure coding practices, avoid common pitfalls, and follow industry-recognized security patterns to reduce the likelihood of vulnerabilities in the final product. |

### 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** | **Correct Use of Data Types** |

| **Noncompliant Code** |
| --- |
| Implicit type conversion can cause data loss or unexpected behavior. Assigning a float to an int results in truncation of the fractional part. |
| int a = 10;  float b = 3.14;  a = b; // Implicit conversion between incompatible types |

| **Compliant Code** |
| --- |
| We ensure no unintended data loss or behavior change by explicitly casting the float to int. |
| float a = 10.5;  int b = static\_cast<int>(a); // Explicit type conversion |

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

| **Principles(s):**   1. Validate Input Data 2. Heed Compiler Warnings 3. Keep It Simple 4. Default Deny 5. Use Effective Quality Assurance Techniques   **Justification**:   1. **Validate Input Data:** Ensuring that data is validated helps prevent implicit conversions that may lead to unexpected behavior or data loss. 2. **Heed Compiler Warnings:** Compiler warnings about type conversions should be addressed to prevent potential bugs and vulnerabilities. 3. **Keep It Simple**: Explicit type conversion simplifies the code and makes it easier to understand, reducing the chance of errors. 4. **Default Deny:** Implicit type conversions should be prevented unless explicitly allowed to minimize unexpected outcomes. 5. **Use Effective Quality Assurance Techniques**: Quality assurance techniques, like static analysis, can catch improper type conversions early in the development process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0.0 | clang-analyzer-core  **Link**: [clang-tidy - Clang-Tidy Checks — Extra Clang Tools 20.0.0git documentation](https://clang.llvm.org/extra/clang-tidy/checks/list.html?form=MG0AV3) | This tool performs static analysis on C++ code and checks for various issues, including implicit type conversions that could lead to data loss or unexpected behavior. The clang-analyzer-core checker can identify risky type conversions and provide recommendations for explicit casting. |
| SonarQube | 9.6 | cpp:S1166 – "Primitive type is not used with a cast".  **Link**: [C/C++/Objective-C](https://docs.sonarsource.com/sonarqube-server/10.5/analyzing-source-code/languages/c-family/?form=MG0AV3) | SonarQube’s C++ analyzer detects implicit type casting and ensures that proper casting is used to prevent data loss. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | **STD-002-CPP** | Validate Data Values |

| **Noncompliant Code** |
| --- |
| Directly taking user input without validation allows invalid or malicious data, such as negative numbers or excessively large values. |
| int age;  std::cin >> age; // No validation on input |

| **Compliant Code** |
| --- |
| Validating the input ensures that the age value is within a reasonable range. |
| int age;  std::cin >> age;  if (age < 0 || age > 120) {      std::cerr << "Invalid age input." << std::endl;  } |

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

| **Principles(s):**   1. Validate Input Data 2. Default Deny 3. Sanitize Data Sent to Other Systems   **Justification:**   1. **Validate Input Data**: This principle directly supports the standard by ensuring that all input data is valid and correct before processing. Unchecked inputs can lead to vulnerabilities and erroneous behavior. By validating the input data, we ensure that only data meeting specific criteria is accepted, preventing invalid or malicious data from causing issues. 2. **Default Deny**: This principle supports the standard by treating all input as untrusted unless validated. By default, inputs are assumed to be potentially harmful until proven otherwise through validation. Applying this principle means that unless the input data is explicitly validated and deemed safe, it should not be processed, aligning with the standard's goal of preventing invalid data. 3. **Sanitize Data Sent to Other Systems**: While this principle mainly addresses data being sent to other systems, it is relevant here because validating input data is a form of sanitization. Ensuring that data is clean and safe before use prevents injection attacks and other security issues. By validating input data, we sanitize it before it can affect other parts of the system or be transmitted elsewhere, ensuring comprehensive data integrity and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0.0 | modernize-use-nullptr  **Link**: [clang-tidy - Clang-Tidy Checks — Extra Clang Tools 20.0.0git documentation](https://clang.llvm.org/extra/clang-tidy/checks/list.html?form=MG0AV3) | While primarily focused on modernizing code, Clang-Tidy can also help identify and refactor parts of the code that deal with data validation and type safety. |
| Cppcheck | 2.6 | misra-cpp2008-7-2-1  **Link**: [Cppcheck manual](https://cppcheck.sourceforge.io/manual.pdf?form=MG0AV3) | Cppcheck includes checks for compliance with the MISRA C++ standard, including data validation and input handling rules. |
| SonarQube | 9.6 | cpp:S914 – "Parameter validation should be used"  **Link**: [C/C++/Objective-C](https://docs.sonarsource.com/sonarqube-server/10.5/analyzing-source-code/languages/c-family/?form=MG0AV3) | SonarQube’s C++ analyzer can detect missing input validation and recommend where validation checks should be added. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | **STD-003-CPP** | Proper Handling of Strings |

| **Noncompliant Code** |
| --- |
| The string exceeds the buffer size, causing a buffer overflow, which can lead to memory corruption or security vulnerabilities. |
| char buffer[10];  strcpy(buffer, "This is a very long string"); // Buffer overflow |

| **Compliant Code** |
| --- |
| Using strncpy ensures that the string is properly bounded and null-terminated, avoiding buffer overflows. |
| char buffer[20];  strncpy(buffer, "This is a very long string", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination |

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

| **Principles(s):**   1. **Validate Input Data** 2. **Keep It Simple** 3. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Validate Input Data**: Validating input data ensures that any strings being copied or manipulated do not exceed buffer sizes, preventing buffer overflows. By validating string inputs before operations like copying, the risk of buffer overflow is minimized, aligning with the standard's goal of proper string handling. 2. **Keep It Simple**: Simplifying code makes it easier to identify and prevent potential issues like buffer overflows. Clear and straightforward string handling reduces complexity and enhances security. Using functions like strncpy that inherently support safer string operations promotes simplicity and helps ensure proper string handling. 3. **Use Effective Quality Assurance Techniques**: Quality assurance techniques, such as static analysis and dynamic testing, can detect issues with string handling, ensuring that buffers are properly managed. Implementing QA practices ensures that string handling in the code is tested for edge cases and vulnerabilities, aligning with the standard to prevent buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0.0 | cppcoreguidelines-pro-bounds-array-to-pointer-decay  **Link:** [**clang-tidy - Clang-Tidy Checks — Extra Clang Tools 20.0.0git documentation**](https://clang.llvm.org/extra/clang-tidy/checks/list.html?form=MG0AV3) | Ensures proper handling of C-style strings and prevents buffer overflows. |
| Cppcheck | 2.6 | bufferOverrun  **Link**: [Cppcheck - A tool for static C/C++ code analysis](https://cppcheck.sourceforge.io/?form=MG0AV3) | Checks for buffer overflows by analyzing the bounds of arrays and strings. |
| Coverity | 2023.2 | CWE-120 – Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')  **Link**: [Coverity Scan - Static Analysis](https://scan.coverity.com/?form=MG0AV3) | Static analysis tool that identifies and prevents buffer overflows by checking the size of buffers. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | **STD-004-CPP** | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| This approach directly embeds user input into the SQL query, opening the door for SQL injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| Using prepared statements with parameterized queries ensures user input is treated safely and not executed as part of the SQL query. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  pstmt = conn->prepareStatement(query);  pstmt->setString(1, username); |

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

| **Principles(s):**   1. **Validate Input Data** 2. **Sanitize Data Sent to Other Systems** 3. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Validate Input Data**: Ensuring that input data meets expected formats and constraints before processing helps prevent injection attacks. In the context of SQL, this means validating and sanitizing inputs to ensure they do not contain malicious SQL code. By validating user inputs, the system can prevent malicious data from being embedded in SQL queries, which aligns with the intention to prevent SQL injection. 2. **Sanitize Data Sent to Other Systems**: This principle involves ensuring that data sent to other systems, such as a database, is clean and safe. This prevents injection attacks and other vulnerabilities. Using prepared statements and parameterized queries is a form of sanitizing data before it is sent to the database, ensuring that user input is treated safely and not executed as part of the SQL command. 3. **Use Effective Quality Assurance Techniques**: Comprehensive testing practices, such as static and dynamic analysis, can identify potential SQL injection vulnerabilities before deployment. Implementing QA techniques ensures that SQL queries are tested for injection vulnerabilities, helping to catch and fix issues early in the development process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.6 | cpp:S3649 – "SQL queries should not be vulnerable to injection attacks"  **Link:** [**Security-related rules**](https://docs.sonarsource.com/sonarqube-server/9.6/user-guide/rules/security-related-rules/?form=MG0AV3) | Identifies SQL injection vulnerabilities and suggests using parameterized queries. |
| OWASP ZAP | 2.11.1 | SQL Injection Scan Rule  **Link:** [**ZAP – SQL Injection**](https://www.zaproxy.org/docs/alerts/40018/?form=MG0AV3) | Automated penetration testing tool that identifies SQL injection vulnerabilities in web applications. |
| Checkmarx | 9.0 | SQL Injection Detection  **Link:** [**What is SQL Injection + Examples**](https://checkmarx.com/glossary/sql-injection/?form=MG0AV3) | Static analysis tool that identifies SQL injection vulnerabilities in the codebase. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | **STD-005-CPP** | Use Memory Protection Mechanisms |

| **Noncompliant Code** |
| --- |
| After deleting the memory, the pointer is not reset, leaving a dangling pointer that could lead to undefined behavior. |
| int\* ptr = new int[10];  delete[] ptr;  // ptr is not reset, leading to a dangling pointer |

| **Compliant Code** |
| --- |
| By setting the pointer to nullptr, we ensure that any future access to the pointer does not result in undefined behavior. |
| int\* ptr = new int[10];  delete[] ptr;  ptr = nullptr; // Safe to avoid dangling pointer |

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

| **Principles(s):**   1. **Architect and Design for Security Policies** 2. **Keep It Simple** 3. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Architect and Design for Security Policies**: Designing and implementing secure memory management practices is a key aspect of secure software architecture. This includes ensuring that pointers are properly managed and that any potential security issues, such as dangling pointers, are addressed. By setting pointers to nullptr after deleting memory, we follow a secure design practice that prevents dangling pointers and related security vulnerabilities. 2. **Keep It Simple**: Simple and clear memory management practices, such as setting pointers to nullptr after deletion, reduce the likelihood of errors and make the code easier to understand and maintain. Using straightforward and consistent practices for managing memory aligns with the principle of simplicity, making the code less prone to bugs and security issues. 3. **Use Effective Quality Assurance Techniques**: QA techniques, including static analysis and dynamic testing, can identify issues with memory management, such as uninitialized pointers or memory leaks, ensuring that the code is robust and secure. Implementing QA practices ensures that memory protection mechanisms are validated and tested, helping to catch potential issues early in the development process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | memcheck | Detects memory leaks, uninitialized memory, and invalid memory access. |
| Coverity | 2023.2 | CWE-787 – Out-of-bounds Write | Static analysis tool that identifies out-of-bounds memory accesses. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | **STD-006-CPP** | Use Assertions to Validate Assumptions |

| **Noncompliant Code** |
| --- |
| Using simple if checks is not as efficient or clear as assertions, which automatically halts execution when conditions are violated. |
| if (x < 0) {      std::cerr << "Invalid value!" << std::endl;  } |

| **Compliant Code** |
| --- |
| Assertions indicate programmer assumptions and halt execution when they are violated, aiding in the early detection of logic errors. |
| assert(x >= 0); // Assertion to verify non-negative x |

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

| **Principles(s):**   1. **Architect and Design for Security Policies** 2. **Keep It Simple** 3. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Architect and Design for Security Policies**: Using assertions helps enforce security policies by verifying that critical assumptions hold true at runtime. This ensures that the code behaves as expected and any violations are caught early. Assertions act as safety checks that validate key assumptions, helping to prevent security issues and logical errors by stopping execution if assumptions are violated. 2. **Keep It Simple**: Assertions simplify code by providing clear and direct checks for assumptions. This makes the code easier to read, understand, and maintain, reducing the likelihood of bugs and logical errors. By using assertions, developers can simplify error handling and validation logic, ensuring that the code remains concise and focused on core functionality. 3. **Use Effective Quality Assurance Techniques**: Quality assurance techniques, such as code reviews and testing, benefit from assertions because they provide explicit checks for expected conditions. This helps identify issues early in the development process. Implementing assertions supports QA efforts by automatically verifying assumptions and catching violations during testing, leading to more robust and reliable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0.0 | cppcoreguidelines-assert | Ensures that assumptions in the code are validated using assertions. |
| Cppcheck | 2.6 | assertUsage | Checks for the proper use of assertions in the code. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | **STD-007-CPP** | Handle Exceptions Properly |

| **Noncompliant Code** |
| --- |
| Catching all exceptions without properly identifying the type leads to obscure error handling and can hide serious issues. |
| try {      int a = 10;      int b = 0;      int c = a / b; // Divide by zero error  }  catch (...) {      std::cout << "Error occurred" << std::endl;  } |

| **Compliant Code** |
| --- |
| By catching specific exceptions and providing clear error messages, we make the error handling more transparent and easier to debug. |
| try {      int a = 10;      int b = 0;      if (b == 0) throw std::logic\_error("Division by zero");      int c = a / b;  }  catch (const std::logic\_error& e) {      std::cerr << "Caught exception: " << e.what() << std::endl;  } |

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

| **Principles(s):**   1. **Architect and Design for Security Policies** 2. **Keep It Simple** 3. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Architect and Design for Security Policies**: Proper exception handling is an integral part of secure software architecture. By catching specific exceptions and providing clear error messages, the code is designed to handle errors gracefully and securely, without revealing sensitive information or causing the application to crash unexpectedly. This principle supports the standard by ensuring that exceptions are handled in a way that upholds the security and stability of the application. 2. **Keep It Simple**: Simplifying exception handling by catching specific exceptions and providing clear, concise error messages makes the code more understandable and easier to maintain. This reduces the likelihood of introducing bugs or security vulnerabilities. This principle aligns with the standard by advocating for straightforward and effective error handling practices. 3. **Use Effective Quality Assurance Techniques**: Quality assurance techniques, such as code reviews and testing, benefit from clear and specific exception handling. By identifying and testing for specific exceptions, QA processes can ensure that the application handles errors correctly and robustly. This principle supports the standard by ensuring that exception handling is tested and validated, improving the overall reliability and security of the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0.0 | cppcoreguidelines-avoid-catch-all | Ensures that exceptions are handled properly by catching specific exceptions rather than using catch-all handlers. |
| SonarQube | 9.6 | cpp:S2737 – "Catch specific exceptions instead of using catch-all handlers"  **Link:** [C/C++/Objective-C](https://docs.sonarsource.com/sonarqube-server/10.5/analyzing-source-code/languages/c-family/?form=MG0AV3) | Identifies and suggests improvements for exception handling practices. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **File Handling** | **STD-008-CPP** | Secure File Handling |

| **Noncompliant Code** |
| --- |
| Directly opening files without proper permissions or validation can lead to security risks. |
| FILE\* f = fopen("sensitive\_data.txt", "w");  fwrite(data, sizeof(char), strlen(data), f); // No validation or file permissions check |

| **Compliant Code** |
| --- |
| Proper file permission checks and input sanitization prevent unauthorized access or modification. Additionally, handling the case where the file cannot be opened and ensuring the file is closed properly are important for robust code. |
| FILE\* f = fopen("sensitive\_data.txt", "w");  if (f) {  // Check file permissions and sanitize data before writing  fwrite(data, sizeof(char), strlen(data), f);  fclose(f); // Don't forget to close the file  } else {  // Handle the error if the file cannot be opened  std::cerr << "Failed to open file!" << std::endl;  } |

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

| **Principles(s):**   1. **Validate Input Data** 2. **Architect and Design for Security Policies** 3. **Default Deny** 4. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Validate Input Data**: Ensuring that input data, such as file names and data to be written, is validated before use helps prevent security vulnerabilities, such as path traversal and injection attacks. Validating and sanitizing file names and data before opening files or writing to them ensures that only trusted and safe data is processed, aligning with the standard for secure file handling. 2. **Architect and Design for Security Policies**: Secure file handling should be embedded into the design and architecture of the software, including proper permissions and validation checks. Designing with security in mind ensures that file operations are performed safely and that potential vulnerabilities are mitigated from the outset, supporting the standard for secure file handling. 3. **Default Deny**: This principle implies that all file operations should be denied by default unless explicitly allowed. This reduces the attack surface by ensuring that only authorized operations are performed. By default, denying file access unless proper permissions are checked and validated helps prevent unauthorized access and modification, aligning with the standard for secure file handling. 4. **Use Effective Quality Assurance Techniques**: QA techniques, including static analysis, code reviews, and testing, help identify potential issues with file handling, ensuring that the code is robust and secure. Implementing QA practices ensures that file handling operations are tested for vulnerabilities and errors, improving the overall security and reliability of the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0.0 | cppcoreguidelines-avoid-magic-numbers | Ensures proper file handling practices and avoids the use of hard-coded values for file operations. |
| Cppcheck | 2.6 | fileAccess | Checks for proper file handling practices, including file permission checks and validation. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Allocation** | **STD-009-CPP** | Secure Memory Allocation |

| **Noncompliant Code** |
| --- |
| Accessing memory beyond the allocated range causes buffer overflow and potential memory corruption. |
| char\* buffer = new char[1024];  buffer[1024] = 'A'; // Out-of-bounds access |

| **Compliant Code** |
| --- |
| Ensuring that memory allocation accounts for null terminators and other bounds improves security. |
| char\* buffer = new char[1025]; // Allocate space for 1024 bytes + null terminator  buffer[1024] = 'A'; // Safe access within bounds |

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

| **Principles(s):**   1. **Validate Input Data** 2. **Architect and Design for Security Policies** 3. **Keep It Simple** 4. **Use Effective Quality Assurance Techniques**   **Justification:**   1. **Validate Input Data**: Validating the size of memory allocations and ensuring that buffers are appropriately sized for their intended use prevents overflows and other memory-related vulnerabilities. By validating the requirements for memory allocation, we ensure that sufficient memory is allocated to handle the data, preventing out-of-bounds access and buffer overflows. 2. **Architect and Design for Security Policies**: Secure memory allocation practices should be integrated into the architecture and design of the software. This includes considering security implications during the design phase and ensuring that memory allocation is handled safely. Designing the software with secure memory management in mind ensures that memory-related vulnerabilities are mitigated from the outset, supporting the standard for secure memory allocation. 3. **Keep It Simple**: Simplifying memory management practices, such as ensuring proper buffer sizes and clear allocation logic, reduces the likelihood of errors and makes the code easier to understand and maintain. By keeping memory allocation practices simple and clear, we minimize the risk of introducing buffer overflows and related vulnerabilities, aligning with the standard for secure memory allocation. 4. **Use Effective Quality Assurance Techniques**: QA techniques, such as static analysis, dynamic testing, and code reviews, can identify potential issues with memory allocation, ensuring that the code is robust and secure. Implementing QA practices ensures that memory allocations are tested for correctness and safety, catching potential vulnerabilities early in the development process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | memcheck | Detects issues with memory allocation, including memory leaks and invalid memory access. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Networking Coding** | **STD-010-CPP** | Secure Networking Practices |

| **Noncompliant Code** |
| --- |
| Unencrypted communications are susceptible to interception, compromising data security. |
| // Using unencrypted sockets  connect(socket, addr, sizeof(addr)); // No encryption or verification |

| **Compliant Code** |
| --- |
| Using secure protocols like TLS ensures that data sent over the network is encrypted and protected. |
| // Using TLS for secure communication  SSL\* ssl\_socket = SSL\_new(ctx);  SSL\_connect(ssl\_socket); |

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

| **Principles(s):**   1. **Architect and Design for Security Policies** 2. **Default Deny** 3. **Sanitize Data Sent to Other Systems** 4. **Practice Defense in Depth**   **Justification:**   1. **Architect and Design for Security Policies**: Implementing secure networking practices requires designing the system with security policies in mind, such as using encrypted communication channels to protect data integrity and confidentiality. By designing the system to use secure protocols like TLS, we ensure that data transmitted over the network is encrypted and protected against interception, aligning with the standard for secure networking practices. 2. **Default Deny**: The principle of default deny implies that all network communications should be considered untrusted and potentially malicious unless explicitly secured. Only secure and verified connections should be allowed. Using secure protocols and verifying connections ensures that only authorized and secure communications are permitted, reducing the risk of unauthorized data access. 3. **Sanitize Data Sent to Other Systems**: When transmitting data over the network, it is crucial to ensure that the data is sanitized and securely transmitted to prevent interception and manipulation. Using TLS for secure communication ensures that data is encrypted and protected, preventing it from being intercepted or tampered with during transmission. 4. **Practice Defense in Depth**: Defense in depth involves implementing multiple layers of security to protect data. Secure networking practices, such as using encryption, are an essential part of this strategy. By using TLS and other secure protocols, we add a layer of protection to the data transmission process, ensuring that even if one layer is breached, the data remains secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Netcat | N/A | Secure Network Communication | Netcat is a versatile networking utility that uses TCP or UDP to read from and write to network connections. It can be used for various network-related tasks, such as port scanning, transferring files, and monitoring network traffic, making it a valuable tool for identifying and troubleshooting network vulnerabilities. |
| OWASP ZAP | 2.11.1 | Secure Network Communication | Automated penetration testing tool that ensures secure networking practices by identifying vulnerabilities in network communications. |

### 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.

**Integration of Automation in the DevSecOps Process**

Green Pace already has a well-established DevOps process and infrastructure. To ensure the enforcement and compliance of the security standards defined in this policy, it is essential to integrate automation at various stages of the DevSecOps pipeline. Automation will help in consistently applying security checks, reducing manual effort, and quickly identifying and mitigating vulnerabilities. The following paragraphs summarize how and where automation should be implemented in the existing DevOps process, guided by the DevSecOps toolchain diagram.

**Create Stage:**

At this stage, developers write and commit code. Integrating security plug-ins within the Integrated Development Environment (IDE) can provide real-time feedback and code analysis to developers. These plug-ins will help identify and fix security issues early in the development process. Tools like static code analyzers (e.g., SonarQube) can automatically scan the code for compliance with coding standards and highlight any vulnerabilities.

**Verify Stage:**

In this stage, the code is built and tested. Implementing Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), Interactive Application Security Testing (IAST), and Software Composition Analysis (SCA) tools can automate the scanning of code for vulnerabilities. These tools can ensure that the code adheres to the security standards and detect any potential security flaws before the code progresses further in the pipeline.

**Preprod Stage:**

Before deployment, the pre-production environment should simulate the production environment as closely as possible. Tools like Chaos Monkey for resilience testing, input fuzzing for identifying unexpected input handling issues, and automated integration tests can ensure that security standards are maintained throughout the integration process. These tools can validate the robustness and security of the application in a controlled environment.

**Plan Stage:**

Automation can be used to track and manage technical security debt, generate DevSec metrics, and incorporate threat modeling tools. Providing automated training modules for security tools can keep the team updated on the latest security practices. This stage ensures that security considerations are integrated into the planning and design phases of the project.

**Release Stage:**

Automating the process of software signing ensures the integrity and authenticity of the code being released. This step is crucial in verifying that the code has not been tampered with and is from a trusted source.

**Prevent Stage:**

Implementing automated tools for signature verification, integrity checks, and other defense-in-depth measures can help prevent security breaches. These tools can continuously monitor for any anomalies and ensure that preventive measures are in place.

**Detect Stage:**

Using Runtime Application Self-Protection (RASP), User and Entity Behavior Analytics (UEBA), and automated network monitoring tools can detect potential security threats. Regular automated penetration tests should also be conducted to identify vulnerabilities and ensure continuous security compliance.

**Respond Stage:**

Automating security orchestration processes can streamline incident response. Implementing RASP and Web Application Firewall (WAF) shielding, along with code obfuscation techniques, can provide immediate protection against attacks and minimize the impact of security incidents.

**Predict Stage:**

Utilizing tools for correlated vulnerability analysis and integrating Indicators of Compromise (IoC) and Threat Intelligence (TI) feeds using standards like STIX and TAXII can help predict and mitigate potential threats. These tools can provide insights into emerging threats and vulnerabilities, allowing for proactive security measures.

**Adapt Stage:**

Continuous adaptation and updating of security processes are essential. Addressing technical security debt, modifying incident response plans, and updating Do Not Disturb (DND) policies based on the latest threat landscape ensure that the security posture remains robust and up-to-date.

By integrating these automated tools and processes at the specified stages, Green Pace can ensure that security standards are enforced and compliance is maintained throughout the DevSecOps pipeline. This approach not only enhances security but also improves efficiency and consistency in applying security measures.

### 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 | Medium | High | Important |
| STD-002-CPP | High | High | Medium | High | Critical |
| STD-003-CPP | High | High | Medium | High | Critical |
| STD-004-CPP | Critical | High | High | High | Critical |
| STD-005-CPP | High | Medium | Medium | High | Important |
| STD-006-CPP | Medium | Medium | Low | Medium | Moderate |
| STD-007-CPP | Medium | Medium | Low | Medium | Moderate |
| STD-008-CPP | High | High | Medium | High | Critical |
| STD-009-CPP | High | Medium | Medium | High | Important |
| STD-010-CPP | Critical | High | High | High | Critical |

### 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 | **What It Is**: Encryption at rest refers to the process of encrypting data while it is stored on a device or media, such as hard drives, databases, or cloud storage. The data is encrypted to protect it from unauthorized access, even if the storage device is physically compromised.  **How It Is Used**: Strong encryption algorithms, such as Advanced Encryption Standard (AES), are employed to encrypt data before it is written to storage. The encryption keys are securely managed to ensure that only authorized users or systems can decrypt and access the data.  **Why the Policy Applies**: The policy ensures that sensitive data remains secure even when it is not actively being accessed or processed. It applies to all stored data to protect it from theft, loss, or unauthorized access, which could result in data breaches and compromise sensitive information.  **Policy:** All data stored on servers, databases, and storage devices must be encrypted using strong encryption algorithms such as Advanced Encryption Standard (AES). |
| Encryption in flight | **What It Is**: Encryption in flight refers to the process of encrypting data while it is being transmitted over a network. This ensures that the data is protected from eavesdropping, interception, and tampering during transmission.  **How It Is Used**: Secure communication protocols, such as Transport Layer Security (TLS) or Secure Sockets Layer (SSL), are used to encrypt data packets before they are sent over the network. This ensures that the data remains confidential and integral during transmission.  **Why the Policy Applies**: The policy protects data during transit from unauthorized access and interception. It applies to all data transmissions, including web browsing, email, and data transfers between servers, to ensure that sensitive information is not compromised while being sent over the network.  **Policy**: All data transmitted over networks must be encrypted using secure protocols such as Transport Layer Security (TLS) or Secure Sockets Layer (SSL). |
| Encryption in use | **What It Is**: Encryption in use refers to the process of encrypting data while it is being processed or used by applications. This protects the data from being accessed by unauthorized processes or users during runtime.  **How It Is Used**: Techniques such as memory encryption, secure enclaves, and hardware-based security features (e.g., Intel SGX) are used to protect data in active memory. These methods ensure that sensitive data remains encrypted while being processed by applications.  **Why the Policy Applies**: The policy ensures that sensitive data is protected throughout its entire lifecycle, including when it is in active use. It applies to all applications and processes handling sensitive data to prevent unauthorized access, memory scraping, and side-channel attacks.  **Policy**: Sensitive data must be encrypted while being processed or used by applications, ensuring protection against memory scraping and unauthorized access during runtime. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **What It Is**: Authentication is the process of verifying the identity of a user or system before allowing access to resources. This ensures that only authorized entities can interact with the system.  **How It Is Used**: Strong authentication mechanisms, such as passwords, biometric verification (fingerprints, facial recognition), and multi-factor authentication (MFA), are implemented to verify identities. For systems, digital certificates and authentication tokens are used.  **Why the Policy Applies**: The policy applies to all access points within the system, including logins, API calls, and network access. It is essential to prevent unauthorized access and ensure that only verified identities can access sensitive resources. This enhances security by reducing the risk of identity theft, unauthorized access, and potential breaches.  **Policy**: All users and systems must authenticate their identity before accessing any resources. Strong authentication mechanisms such as multi-factor authentication (MFA) must be used. |
| Authorization | **What It Is**: Authorization is the process of determining what resources and actions an authenticated user or system is allowed to access and perform. It ensures that users and systems can only access resources for which they have explicit permission.  **How It Is Used**: Role-based access control (RBAC) and attribute-based access control (ABAC) are implemented to manage permissions. Access control lists (ACLs) and policies are defined to enforce authorization rules, granting the least privilege necessary for tasks.  **Why the Policy Applies**: The policy ensures that access is granted based on predefined roles and permissions, minimizing the risk of unauthorized actions. It applies whenever a user or system requests access to resources, including files, applications, and network services, to prevent unauthorized access and protect sensitive data.  **Policy**: Access to resources must be controlled based on user roles and permissions. The principle of least privilege should be enforced, granting only the necessary access for users to perform their tasks. |
| Accounting | **What It Is**: Accounting, or auditing, is the process of tracking and recording all access and actions taken by users and systems within a system. This provides a trail of activity that can be used for forensic analysis, compliance, and monitoring.  **How It Is Used**: Logging and monitoring tools are used to capture and store audit logs. Security Information and Event Management (SIEM) systems analyze and review logs for any suspicious activity. Regular audits and reviews are conducted to ensure compliance and detect anomalies.  **Why the Policy Applies**: The policy ensures accountability and traceability by recording all access and actions. It is essential for maintaining accountability, compliance with regulations, and detecting security incidents. This policy applies continuously, ensuring that all access and actions are recorded and monitored.  **Policy**: All access and actions must be logged and monitored to ensure accountability and traceability. Audit logs should be maintained securely and reviewed regularly for any suspicious activity. |

**\***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 | 12/8/2024 | Milestone Three | Joshua Pugh |  |
| 1.2 | 12/8/2024 | Project One | Joshua Pugh |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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