**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 | Ensure that all input data, regardless of source, is validated for type, length, format, and range. This reduces vulnerabilities such as SQL injection, cross-site scripting, and buffer overflows, by preventing malicious data from being processed by the system. Proper validation acts as a first line of defense, helping to ensure data integrity and security. |
| 1. Heed Compiler Warnings | Compiler warnings are often indicative of potential security or functional issues in code. Treating these warnings as errors and resolving them can preempt runtime errors and security breaches. Addressing compiler warnings during development fosters cleaner, more secure code, minimizing the risk of exploitable vulnerabilities in production environments. |
| 1. Architect and Design for Security Policies | Incorporate security considerations into the architecture and design phase of software development. By aligning system architecture with security policies, the software is built on a foundation that prioritizes security, ensuring that protective measures are integral and not afterthoughts. This approach facilitates the development of secure applications from the ground up. |
| 1. Keep It Simple | Complexity is the enemy of security. A simple design minimizes the chances of error and misconfiguration, which can lead to security vulnerabilities. By maintaining simplicity in code and architecture, developers can better understand, maintain, and secure the system, making it more resilient against attacks. |
| 1. Default Deny | Adopt a security stance where access is denied by default, and permissions are granted only when absolutely necessary. This principle reduces the potential attack surface by ensuring that unless explicitly allowed, actions are considered unauthorized, thereby enhancing the overall security posture of systems. |
| 1. Adhere to the Principle of Least Privilege | Each component of the system, be it a process, user, or program, should operate using the minimal set of privileges necessary to complete its tasks. This limits the damage that can be done in the event of a compromise, as malicious actors gain access only to minimal rights and resources. |
| 1. Sanitize Data Sent to Other Systems | Cleanse data before it is sent to other systems to prevent inadvertent leakage of sensitive information or the propagation of malicious content. This practice is crucial for maintaining the integrity and security of data as it moves between systems, platforms, or services. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls throughout the IT environment. This strategy ensures that if one layer fails, others still provide protection, creating a more robust security posture that is resilient against various types of attacks. |
| 1. Use Effective Quality Assurance Techniques | Deploy comprehensive quality assurance (QA) techniques that encompass security testing to identify and mitigate vulnerabilities before software release. Effective QA practices are vital for confirming the security and functionality of software, thus preventing defects that could be exploited by attackers. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards that are designed to prevent common security vulnerabilities. By adhering to these standards, developers can reduce the likelihood of introducing security flaws into their code, making it harder for attackers to exploit the application. |

### 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 | Enforces correct usage of data types to prevent type-related errors and vulnerabilities, enhancing the reliability and security of the software. |

| **Noncompliant Code** |
| --- |
| Implicit conversion from long to int can cause data loss which might lead to undefined behavior or security flaws. |
| int i;  long l = 10000000000;  i = l; // implicit conversion might cause data loss |

| **Compliant Code** |
| --- |
| Using static\_cast makes the type conversion explicit, preventing unintended data loss. |
| long l = 10000000000;  int i = static\_cast<int>(l); // explicit conversion clarifies the developer's intent |

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

| **Principles(s):**  Validate Input Data: This principle ensures that all input data, regardless of the source, is validated for type, length, format, and range before being processed. This is crucial to prevent attacks such as SQL injection, cross-site scripting (XSS), and buffer overflows. Validation acts as a first line of defense, protecting the system from malicious or erroneous input and ensuring the data's integrity.  Heed Compiler Warnings: Compiler warnings often indicate potential security or functional issues in the code. Ignoring these warnings can lead to runtime errors and security vulnerabilities. Treating compiler warnings as errors forces developers to address potential issues early in the development process, improving the overall security and stability of the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.1 | C++ Type Safety Rules | Detects implicit conversions and potential data loss from incorrect data type usage. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensures that data values are within defined safe limits, thus preventing errors like buffer overflows or incorrect data processing. |

| **Noncompliant Code** |
| --- |
| The code does not verify if index is within the bounds of the array, leading to potential buffer overflow. |
| int index = userInput();  array[index] = 10; // No check if index is within the bounds of the array |

| **Compliant Code** |
| --- |
| The index is checked to ensure it lies within valid array bounds, preventing buffer overflow. |
| int index = userInput();  if (index >= 0 && index < array\_size) {  array[index] = 10;  } |

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

| **Principles(s):**  Validate Input Data: This principle ensures that all input data, regardless of the source, is validated for type, length, format, and range before being processed. This is crucial to prevent attacks such as SQL injection, cross-site scripting (XSS), and buffer overflows. Validation acts as a first line of defense, protecting the system from malicious or erroneous input and ensuring the data's integrity.  Default Deny: The default deny principle ensures that, by default, access to system resources is denied unless explicitly allowed. This limits the attack surface by ensuring that no unintended access is granted. This principle is a fundamental aspect of least privilege, where only the minimum necessary permissions are granted to users or processes.. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.12 | BoundsChecker | Automatically detects array bounds violations and buffer overflow vulnerabilities. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Prevents common string-related vulnerabilities such as buffer overflows by ensuring strings are handled correctly. |

| **Noncompliant Code** |
| --- |
| The code causes a buffer overflow by copying a string that is longer than the allocated buffer. |
| char \*s = new char[10];  strcpy(s, "A very long string"); |

| **Compliant Code** |
| --- |
| The buffer size is appropriate, and strncpy is used with explicit length to prevent overflow. |
| char \*s = new char[20];  strncpy(s, "A very long string", 19);  s[19] = '\0'; // Ensuring null termination |

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

| **Principles(s):**  Sanitize Data Sent to Other Systems: Before sending data to other systems, it should be properly sanitized and cleansed. This principle prevents the unintentional transmission of sensitive information and the propagation of malicious content. Proper sanitization ensures that data leaving the system does not contain malware, confidential information, or improperly formatted content.  Validate Input Data: This principle ensures that all input data, regardless of the source, is validated for type, length, format, and range before being processed. This is crucial to prevent attacks such as SQL injection, cross-site scripting (XSS), and buffer overflows. Validation acts as a first line of defense, protecting the system from malicious or erroneous input and ensuring the data's integrity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 20.2 | Buffer Overflow Detection | Scans for buffer overflow vulnerabilities in string operations. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Secures applications from SQL injection, which can lead to unauthorized data access or data corruption. |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into an SQL query can lead to SQL injection attacks. |
| std::string query = "SELECT \* FROM users WHERE user\_id = " + user\_input;  executeQuery(query); |

| **Compliant Code** |
| --- |
| Uses prepared statements with parameterized queries to safely include user input. |
| std::string query = "SELECT \* FROM users WHERE user\_id = ?";  PreparedStatement \*stmt = conn->prepareStatement(query);  stmt->setString(1, user\_input);  executeQuery(stmt); |

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

| **Principles(s):**  Validate Input Data: This principle ensures that all input data, regardless of the source, is validated for type, length, format, and range before being processed. This is crucial to prevent attacks such as SQL injection, cross-site scripting (XSS), and buffer overflows. Validation acts as a first line of defense, protecting the system from malicious or erroneous input and ensuring the data's integrity.  Sanitize Data Sent to Other Systems: Before sending data to other systems, it should be properly sanitized and cleansed. This principle prevents the unintentional transmission of sensitive information and the propagation of malicious content. Proper sanitization ensures that data leaving the system does not contain malware, confidential information, or improperly formatted content. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.1 | SQL Injection Detection | Detects SQL injection vulnerabilities caused by unvalidated user input. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Prevents memory leaks and corruption by ensuring safe and correct memory allocation and deallocation. |

| **Noncompliant Code** |
| --- |
| Incorrect use of delete on an array allocated with new[], which can lead to memory corruption. |
| char \*p = new char[10];  delete p; // incorrect deallocation |

| **Compliant Code** |
| --- |
| Uses delete[] for an array allocated with new[], ensuring correct memory management. |
| char \*p = new char[10];  delete[] p; // correct deallocation |

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

| **Principles(s):**  Adopt a Secure Coding Standard: Secure coding standards help developers write code that avoids common security vulnerabilities. Following these standards ensures that code is written in a way that makes it less prone to flaws that attackers can exploit. Using secure coding guidelines helps prevent common issues such as buffer overflows, race conditions, and injection attacks.  Practice Defense in Depth: Defense in depth is a strategy that uses multiple layers of security controls to protect systems and data. This way, if one layer fails, additional layers provide ongoing protection. These layers could include firewalls, encryption, authentication mechanisms, intrusion detection systems, and more. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.12 | Memory Leak Detection | Scans for improper memory allocation and deallocation, preventing memory leaks and corruption |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Uses assertions to catch bugs during development by verifying assumptions made in the code, thus preventing incorrect operations. |

| **Noncompliant Code** |
| --- |
| The function fails to check for division by zero, which can cause runtime errors. |
| int divide(int x, int y) {  return x / y; // No checks for zero  } |

| **Compliant Code** |
| --- |
| Asserts that the denominator is not zero, preventing division by zero errors. |
| int divide(int x, int y) {  assert(y != 0);  return x / y;  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques: Comprehensive quality assurance (QA) techniques, including security testing, are critical to identifying and mitigating vulnerabilities before software is released. QA techniques should include unit tests, integration tests, static analysis, dynamic analysis, and penetration testing to ensure both the functionality and security of the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | 4.5 | Assertion Rule | Ensures assertions are used for validating critical assumptions made in code. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Ensures that exceptions are handled gracefully, maintaining the stability and security of applications under error conditions. |

| **Noncompliant Code** |
| --- |
| Throws a raw string literal as an exception, which is not recommended. |
| int calculate(int x, int y) {  if (y == 0) throw "Division by zero error";  return x / y;  } |

| **Compliant Code** |
| --- |
| Throws a standard exception type, which is safer and more informative. |
| int calculate(int x, int y) {  if (y == 0) throw std::runtime\_error("Division by zero error");  return x / y;  } |

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

| **Principles(s):**  Keep It Simple: Complexity increases the likelihood of errors, misconfigurations, and vulnerabilities. A simpler design is easier to understand, maintain, and secure. Keeping systems, code, and architecture as simple as possible minimizes the attack surface and reduces the risk of security flaws.  Adhere to the Principle of Least Privilege: This principle limits access rights for users, systems, or processes to the bare minimum necessary to perform their tasks. By minimizing privileges, the potential damage from a compromised account or process is significantly reduced. This is especially important in cases where sensitive data or critical system components are involved. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.1 | Exception Handling | Ensures proper handling of exceptions and prevents poor error management. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety Practices | STD-008-CPP | Ensures that the application behaves correctly under concurrent execution, preventing race conditions and deadlocks. |

| **Noncompliant Code** |
| --- |
| Incrementing a shared variable without synchronization can lead to race conditions. |
| int counter = 0;  void increment() {  counter++; // Not thread-safe  } |

| **Compliant Code** |
| --- |
| Uses a mutex to synchronize access to the shared variable, ensuring thread safety. |
| std::mutex mtx;  int counter = 0;  void increment() {  std::lock\_guard<std::mutex> lock(mtx);  counter++; // Thread-safe  } |

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

| **Principles(s):**  Practice Defense in Depth: Defense in depth is a strategy that uses multiple layers of security controls to protect systems and data. This way, if one layer fails, additional layers provide ongoing protection. These layers could include firewalls, encryption, authentication mechanisms, intrusion detection systems, and more. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.12 | Thread Safety Detection | Detects race conditions and deadlock scenarios, ensuring thread safety in concurrent execution. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Comprehensive Error Handling | STD-009-CPP | Promotes robust software by handling potential errors gracefully, thus maintaining system stability and preventing crashes. |

| **Noncompliant Code** |
| --- |
| Accesses an array element without checking if the index is within the bounds, risking a segmentation fault. |
| int accessArray(int index, int\* array, int size) {  return array[index]; // No bounds checking  } |

| **Compliant Code** |
| --- |
| Checks if the index is within bounds before accessing the array, throwing an exception if not, which prevents out-of-bounds access. |
| int accessArray(int index, int\* array, int size) {  if (index >= 0 && index < size) {  return array[index];  } else {  throw std::out\_of\_range("Index out of range");  }  } |

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

| **Principles(s):**  Sanitize Data Sent to Other Systems: Before sending data to other systems, it should be properly sanitized and cleansed. This principle prevents the unintentional transmission of sensitive information and the propagation of malicious content. Proper sanitization ensures that data leaving the system does not contain malware, confidential information, or improperly formatted content.  Validate Input Data: This principle ensures that all input data, regardless of the source, is validated for type, length, format, and range before being processed. This is crucial to prevent attacks such as SQL injection, cross-site scripting (XSS), and buffer overflows. Validation acts as a first line of defense, protecting the system from malicious or erroneous input and ensuring the data's integrity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.1 | Bounds Checking | Detects array bounds violations and potential segmentation faults due to improper error handling. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Cryptographic Operations | STD-010-CPP | Ensures the confidentiality and integrity of data through the correct implementation of cryptographic algorithms, preventing data breaches. |

| **Noncompliant Code** |
| --- |
| Utilizes MD5 for hashing, which is considered weak and vulnerable to collisions. |
| // Using a weak cryptographic hash function  unsigned char \*hashData(const char \*data) {  return MD5(data, strlen(data), nullptr);  } |

| **Compliant Code** |
| --- |
| Uses SHA-256 for hashing, providing a stronger, more secure cryptographic hash. |
| unsigned char \*hashData(const char \*data) {  return SHA256(data, strlen(data), nullptr); // Using a strong hash function  } |

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

| **Principles(s):**  Adopt a Secure Coding Standard: Secure coding standards help developers write code that avoids common security vulnerabilities. Following these standards ensures that code is written in a way that makes it less prone to flaws that attackers can exploit. Using secure coding guidelines helps prevent common issues such as buffer overflows, race conditions, and injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 20.2 | Weak Hash Detection | Scans for usage of weak cryptographic algorithms like MD5. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

**Plan Phase**: Automate threat modeling and include security metrics as part of the planning.

**Create Phase**: Implement IDE security plugins to automatically detect vulnerabilities during development.

**Verify Phase**: Use SAST/DAST/IAST tools to continuously check the code for security flaws before deployment.

**Release Phase**: Ensure software signing mechanisms are in place to validate the integrity of the release.

### 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 | 4 |
| STD-002-CPP | High | Medium | Medium | High | 4 |
| STD-003-CPP | High | Medium | Medium | High | 4 |
| STD-004-CPP | High | High | Medium | Critical | 5 |
| STD-005-CPP | High | Low | Medium | High | 4 |
| STD-006-CPP | Medium | Low | Low | Medium | 3 |
| STD-007-CPP | Medium | Medium | Low | Medium | 3 |
| STD-008-CPP | High | Low | High | High | 5 |
| STD-009-CPP | Medium | Low | Medium | Medium | 4 |
| STD-010-CPP | High | Medium | High | High | 5 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Protects data stored on disks, databases, or other storage mediums. Applied to all sensitive and personally identifiable data. |
| Encryption in flight | Protects data as it is transmitted across networks. Applied to all data communication between systems. |
| Encryption in use | Protects data currently being processed in memory or used by the system. Applied to sensitive data in active processes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Ensures users are who they claim to be. Applied to all user logins with multifactor authentication. |
| Authorization | Ensures users have permission for actions. Applied to all role-based access controls. |
| Accounting | Logs user activities. Applied to tracking changes, access, and audits. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 10/13/2024 | [Insert text.] | Joshua Donnelly | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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