**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 for ensuring that any data received from external sources, such as user input, conforms to expected formats, types, and constraints. This prevents common vulnerabilities like SQL injection, buffer overflows, and cross-site scripting (XSS). By validating inputs, developers can ensure that the application processes only legitimate and safe data, reducing the risk of security breaches. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential issues in the code, such as type mismatches, deprecated functions, or memory leaks. Ignoring these warnings can leave the code vulnerable to security flaws. Addressing compiler warnings during development helps identify and fix potential vulnerabilities early in the process, improving the overall security and stability of the application. |
| 1. Architect and Design for Security Policies | Security should be considered from the beginning of the software development lifecycle. By integrating security policies into the architecture and design phase, developers can identify potential security issues and implement appropriate controls. This approach ensures that security is not an afterthought but a foundational aspect of the system, reducing the likelihood of vulnerabilities. |
| 1. Keep It Simple | Complex systems are more challenging to understand, implement, and secure. By keeping the design and implementation of the system as simple as possible, developers can minimize the risk of introducing security flaws. Simpler code is easier to review, maintain, and test, making it more resilient to attacks and easier to secure. |
| 1. Default Deny | The principle of default deny, also known as fail-safe defaults, dictates that access to resources should be denied by default unless explicitly granted. This ensures that, in the absence of specific permissions, users and systems cannot access or modify resources. This approach minimizes the risk of unauthorized access and helps enforce strict access control policies. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege requires that users and systems have the minimum level of access necessary to perform their functions. By restricting permissions, the impact of a potential security breach is minimized, as attackers have limited access to critical systems and data. This principle is essential for reducing the attack surface and mitigating the risk of insider threats. |
| 1. Sanitize Data Sent to Other Systems | When data is sent from one system to another, it should be sanitized to remove any potentially harmful content, such as malicious code or unauthorized commands. This practice prevents vulnerabilities like injection attacks and ensures that the receiving system processes only safe and valid data. Proper data sanitization is essential for maintaining the integrity and security of interconnected systems. |
| 1. Practice Defense in Depth | Defense in depth is a multi-layered approach to security, where multiple protective measures are implemented at various levels of the system. This strategy ensures that if one layer is compromised, other layers still provide protection. By employing multiple security controls, such as firewalls, encryption, and intrusion detection systems, organizations can better protect against a wide range of threats. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques, such as code reviews, automated testing, and static analysis, are critical for identifying and addressing security vulnerabilities during the development process. By incorporating these techniques, developers can catch and fix issues before they reach production, enhancing the security and reliability of the software. |
| 1. Adopt a Secure Coding Standard | Secure coding standards provide guidelines and best practices for writing code that is resistant to common vulnerabilities. By adhering to these standards, developers can avoid dangerous coding patterns and ensure that security is consistently addressed throughout the codebase. This practice helps create more robust and secure applications, reducing the likelihood of exploitable flaws. |

### 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-DAT | Using incorrect data types can lead to unintended behavior, such as data corruption, overflow, or security vulnerabilities. Choosing appropriate data types ensures that variables have the correct size and range, which helps prevent bugs and enhances code stability and security. |

| **Noncompliant Code** |
| --- |
| The variable age is incorrectly set as a signed integer, allowing negative values that do not make sense for age. The salary variable uses an unsigned integer with a negative value, leading to unexpected behavior due to overflow. |
| int age = -1; // Using a signed integer to represent age, which should always be positive  unsigned int salary = -5000; // Using an unsigned integer with a negative value, which results in overflow |

| **Compliant Code** |
| --- |
| Compliant Code Descriptions:  The age variable is set as an unsigned integer, preventing negative values. The salary variable is a signed integer, allowing for both positive and negative values without overflow. |
| unsigned int age = 0; // Use an unsigned integer for age, since it cannot be negative  int salary = 5000; // Use a signed integer for salary to represent both positive and negative values |

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

| **Principles(s):** Validate Input Data: This principle maps to the coding standard as ensuring correct data types is a form of input validation. By enforcing proper data types, the program reduces the risk of accepting invalid data, which could lead to vulnerabilities or incorrect logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | modernize-\* | Identifies incorrect data types usage |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-DAT | Ensuring that data values fall within acceptable ranges prevents unexpected behavior, errors, and vulnerabilities. It is important to validate that inputs conform to expected constraints before processing them. |

| **Noncompliant Code** |
| --- |
| The variable percentage is set to 120 without validation, which could lead to errors in logic assuming the value is always between 0 and 100. |
| int percentage = 120; // No check on the range of percentage  if (percentage > 100) {  // Assuming percentage is always between 0 and 100, could lead to logic errors  } |

| **Compliant Code** |
| --- |
| The percentage value is validated to ensure it falls between 0 and 100. If the value is out of range, an error message is displayed, and the program exits gracefully. |
| int percentage = 120;  if (percentage < 0 || percentage > 100) {  std::cerr << "Invalid percentage value!" << std::endl;  return;  } |

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

| **Principles(s):** Validate Input Data: Input values should be validated to ensure they fall within an acceptable range, reducing the risk of logic errors or vulnerabilities due to incorrect values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | cppcoreguidelines-\* | Identifies improper value range validation |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-STR | Unsafe string functions, such as strcpy or strcat, can lead to buffer overflows if the destination buffer is not large enough to hold the source string. Using safer alternatives like strncpy helps prevent such vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Using strcpy without checking the buffer size can result in a buffer overflow, as the source string is longer than the destination buffer. |
| char buffer[10];  strcpy(buffer, "This is too long for the buffer"); // Can cause buffer overflow |

| **Compliant Code** |
| --- |
| strncpy is used to copy only the first sizeof(buffer) - 1 characters, ensuring no buffer overflow occurs. The last character is explicitly set to null to ensure proper string termination. |
| char buffer[10];  strncpy(buffer, "This is too long for the buffer", 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):** Keep It Simple: By using safer alternatives, the complexity of string operations is reduced, making them easier to maintain and secure. Validate Input Data: This principle applies because string inputs need to be checked to ensure they don’t exceed the buffer size. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | clang-analyzer-\* | Identifies unsafe string usage patterns |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-SQL | Using parameterized queries prevents SQL injection attacks by ensuring that user input is treated as data, not executable code. This is a critical defense against one of the most common security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The username input is directly concatenated into the query string, allowing an attacker to inject malicious SQL code, potentially leading to data breaches. |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "';";  // User input directly included in query, vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| A parameterized query is used, which treats username as data rather than part of the query, preventing SQL injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  std::preparedStatement stmt = db.prepareStatement(query);  stmt.setString(1, username);  // Using a parameterized query to prevent SQL injection |

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

| **Principles(s):** Sanitize Data Sent to Other Systems: This principle applies because user input must be sanitized before being used in queries to avoid injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | SQL Injection rule | Detects SQL injection vulnerabilities automatically |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-MEM | Memory leaks occur when dynamically allocated memory is not properly deallocated, leading to wasted resources and potential system instability. Proper memory management is crucial to prevent such issues. |

| **Noncompliant Code** |
| --- |
| Dynamically allocated memory for ptr is not deallocated, causing a memory leak. |
| int\* ptr = new int[10];  // No delete[] called, resulting in a memory leak |

| **Compliant Code** |
| --- |
| The allocated memory is correctly deallocated using delete[], preventing memory leaks. |
| int\* ptr = new int[10];  delete[] ptr; // Properly deallocates memory |

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

| **Principles(s):** Adhere to the Principle of Least Privilege: Proper memory management ensures that resources are allocated and deallocated efficiently, minimizing the risk of misuse or waste. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | memcheck | Detects memory leaks and improper deallocation |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-ASR | Assertions help identify bugs and invalid states during development by validating assumptions. They should be used to catch errors early in the development process, improving code quality and robustness. |

| **Noncompliant Code** |
| --- |
| An invalid state is not checked, which can lead to undefined behavior or crashes if index is used in an array access. |
| int index = -1;  // No check for valid index |

| **Compliant Code** |
| --- |
| The assert statement ensures index is non-negative, catching invalid states during development. |
| int index = -1;  assert(index >= 0); // Ensures index is non-negative |

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

| **Principles(s):** Validate Input Data: Ensuring that assumptions about variables are correct is a form of validation that prevents invalid states from affecting program flow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | clang-analyzer-\* | Detects assertion failures and invalid states |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-EXC | Proper exception handling ensures that programs can gracefully recover from errors or unexpected conditions without crashing. This improves stability and user experience. |

| **Noncompliant Code** |
| --- |
| Catching all exceptions without proper handling can obscure the root cause of errors and make debugging difficult. |
| try {  // Code that may throw an exception  } catch (...) {  // Catching all exceptions without handling, hides issues  } |

| **Compliant Code** |
| --- |
| Specific exceptions are caught and logged, allowing for appropriate handling and easier debugging. |
| try {  // Code that may throw an exception  } catch (const std::exception& e) {  std::cerr << "Exception caught: " << e.what() << std::endl; // Properly handles exceptions  } |

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

| **Principles(s):** Keep It Simple: Properly handling specific exceptions simplifies debugging and improves code maintainability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | clang-analyzer-\* | Detects improper exception handling patterns |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Double-Free Errors | STD-008-MEM | Double-free errors occur when delete is called more than once on the same pointer, leading to undefined behavior and potential program crashes. Proper management of pointers prevents this issue. |

| **Noncompliant Code** |
| --- |
| The pointer ptr is deleted twice, leading to undefined behavior and potential crashes |
| int\* ptr = new int;  delete ptr;  delete ptr; // Double-free error |

| **Compliant Code** |
| --- |
| After deleting ptr, it is set to nullptr, preventing further deletions and avoiding double-free errors. |
| int\* ptr = new int;  delete ptr;  ptr = nullptr; // Prevents double-free by nullifying pointer |

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

| **Principles(s):** Adhere to the Principle of Least Privilege: Managing memory efficiently reduces the risk of unexpected behavior by limiting the ability of code to misuse freed pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | memcheck | Detects improper memory management like double-free |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Signed Integer Overflow | STD-009-DAT | Signed integer overflow occurs when calculations exceed the range of the variable type, leading to unexpected results. Checking for overflow conditions ensures program correctness and security. |

| **Noncompliant Code** |
| --- |
| The addition results in overflow because a is at the maximum value an integer can hold. |
| int a = INT\_MAX;  int b = a + 1; // Causes overflow |

| **Compliant Code** |
| --- |
| The code checks that a is less than INT\_MAX before performing the addition, ensuring no overflow occurs. |
| int a = INT\_MAX;  if (a < INT\_MAX) {  int b = a + 1; // Safely checks for overflow  } |

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

| **Principles(s):** Validate Input Data: This principle applies because preventing overflow requires checking values before performing operations. Keep It Simple: Adding simple checks for overflow helps to avoid unnecessary complexity and prevents errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | clang-analyzer-\* | Detects integer overflow vulnerabilities |

#### 

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Using Vulnerable Functions like gets() | STD-010-STR | Functions like gets() do not check buffer boundaries, leading to buffer overflow vulnerabilities. Using safer alternatives like fgets() mitigates this risk. |

| **Noncompliant Code** |
| --- |
| The gets() function reads input without checking the size of the destination buffer, potentially causing a buffer overflow. |
| The gets() function reads input without checking the size of the destination buffer, potentially causing a buffer overflow. |

| **Compliant Code** |
| --- |
| The fgets() function reads input while respecting buffer boundaries, preventing buffer overflow. |
| char buffer[10];  fgets(buffer, sizeof(buffer), stdin); // Safe alternative to gets() |

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

| **Principles(s):** Keep It Simple: Using a safer function simplifies the handling of input, making the code more secure. Sanitize Data Sent to Other Systems: The safer input functions ensure that data is properly constrained and validated, avoiding vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0.1 | cppcoreguidelines-\* | Detects unsafe string usage patterns like gets() |

### 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 plays a critical role in ensuring compliance with the security standards defined in this policy. Green Pace, with its established DevOps process, can extend its practices by incorporating security throughout the entire software development lifecycle—this is the essence of DevSecOps.

Pre-Production Phase

The Pre-production phase encompasses several key activities where automation can be applied:

1. Assess and Plan: At this stage, automation tools like SonarQube or Clang-Tidy can be used to perform static analysis and identify potential vulnerabilities before the coding phase even begins. Automated threat modeling tools can also help identify potential risks, based on the system architecture and design decisions.
2. Design: During the design phase, automated security testing for best practices (e.g., using OWASP top 10) can be applied. Security tools should review the design for vulnerabilities, ensuring that security is built into the architecture, not added later.
3. Build: As code is being written, real-time security feedback from integrated tools can ensure that developers are following best practices. Tools like Valgrind for memory management or static analysis tools like SonarQube should be integrated into the build process to automatically identify and report issues such as buffer overflows, memory leaks, or unsafe functions.
4. Verify and Test: Automated testing during this phase includes both functional and security-specific tests. Tools that perform vulnerability scanning and compliance checks should be run as part of the continuous integration (CI) pipeline to ensure that any new changes conform to the established security guidelines before deployment.

Production Phase

The Production phase continues with automation ensuring that security remains a focus:

1. Transition and Health Check: Before any deployment into production, automation tools can perform security validation, such as penetration testing, checking for configuration vulnerabilities, and ensuring that security settings are properly implemented. This ensures that the application meets the necessary security standards.
2. Monitor and Detect: Once in production, automated monitoring tools should be used to detect security threats in real-time. Log collection, Security Information and Event Management (SIEM) systems, and intrusion detection can be automated to monitor for unusual activity or threats. Automated alerts can notify the security team when potential breaches or security violations occur.
3. Respond: When a security incident is detected, automated systems can quickly execute predefined actions. For example, they may block suspicious IP addresses, roll back vulnerable changes, or patch known issues. This minimizes the time between detection and response, ensuring that systems are protected.
4. Maintain and Stabilize: Automated assessments against security baselines ensure that the system remains secure over time. This includes regularly scheduled vulnerability scans and checks for outdated software versions or unpatched vulnerabilities. Automation ensures that any deviations from the security baseline are flagged and handled before they become a threat.

DevSecOps Integration

Incorporating automation across both Pre-production and Production phases ensures that security is continuously enforced. Tools like SonarQube for code analysis, Valgrind for memory management, SIEM systems, and intrusion detection systems (IDS) work together to maintain security throughout the development lifecycle. Automation eliminates human error, ensures compliance with security standards, and allows teams to respond more quickly to security incidents. With DevSecOps, Green Pace can maintain a secure, efficient, and compliant development and production environment.

### 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-DAT | Medium | Possible | Low | High | 3 |
| STD-002-DAT | Medium | Possible | Low | High | 3 |
| STD-003-STR | High | Likely | Low | High | 4 |
| STD-004-SQL | High | Likely | Medium | High | 4 |
| STD-005-MEM | Medium | Likely | Low | Medium | 3 |
| STD-006-ASR | Medium | Possible | Low | High | 3 |
| STD-007-EXC | Medium | Likely | Low | Medium | 3 |
| STD-008-MEM | High | Likely | Low | High | 4 |
| STD-009-DAT | High | Possible | Medium | High | 4 |
| STD-010-STR | High | Likely | Low | High | 4 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is used to secure data stored on physical media, such as databases, servers, and cloud storage. By encrypting this data using strong algorithms like AES-256, Green Pace ensures that even if an unauthorized individual gains access to the storage medium, they will not be able to read or modify the data without the encryption key. This policy applies to all systems that store sensitive information, protecting against physical breaches and potential exposure of stored data. |
| Encryption in flight | Encryption in flight protects data while it is being transmitted over networks. Green Pace mandates the use of encryption protocols such as TLS (Transport Layer Security) to secure network communications, including web traffic, emails, API requests, and file transfers. This prevents eavesdropping and unauthorized access during transmission, safeguarding sensitive information from man-in-the-middle attacks and other network-based threats. |
| Encryption in use | Encryption in use addresses the need to protect data while it is actively being processed in memory. By leveraging technologies such as homomorphic encryption and confidential computing environments, Green Pace ensures that sensitive data remains encrypted even while being processed by applications. This policy is particularly important in multi-tenant cloud environments or scenarios where malware could attempt to access unencrypted data in memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that only legitimate users can access systems, with multi-factor authentication (MFA) being a requirement for all users. This policy significantly reduces the risk of unauthorized access, as stolen credentials alone are not sufficient to gain entry to critical systems. |
| Authorization | Authorization governs what authenticated users can do within the system. Green Pace enforces the principle of least privilege, ensuring that users only have access to the resources necessary for their role. Role-based access control (RBAC) is used to prevent users from having excessive permissions, reducing the risk of insider threats and minimizing the potential impact of compromised accounts. |
|  |  |
| Accounting | Accounting provides visibility and traceability into user actions. Green Pace logs all access and activities across its systems, including user logins, file accesses, changes to databases, and administrative actions. These logs are essential for auditing, monitoring, and compliance, providing a detailed trail of activity that can be used to detect and respond to security incidents. Regular review of these logs ensures accountability and helps maintain the integrity of the systems. |

**\***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 | 22SEP2024 | Coding Standards | Lee Kitchen |  |
| 1.2 | 07OCT2024 | Completion of coding standards+ | Lee Kitchen |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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