**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 | Validate all input data to ensure it is both expected and safe. Input validation helps prevent injection attacks and data corruption by ensuring that only properly formatted data enters the system. This principle involves checking data type, length, format, and range. |
| 1. Heed Compiler Warnings | Always pay attention to compiler warnings and resolve them promptly. Compiler warnings can indicate potential security vulnerabilities or logic errors in the code. Ignoring these warnings can lead to exploitable bugs and unstable applications. |
| 1. Architect and Design for Security Policies | Integrate security into the architecture and design phases of development. This principle ensures that security measures are built into the system from the ground up rather than being added as an afterthought. This includes defining clear security policies and ensuring that all components adhere to them. |
| 1. Keep It Simple | **Keep It Simple** Simplify the design and implementation of security measures. Complex systems are more prone to errors and harder to secure. A straightforward and clear design reduces the likelihood of introducing security vulnerabilities and makes the system easier to understand and maintain. |
| 1. Default Deny | Default to denying access unless explicitly allowed. This principle ensures that only authorized users and processes can access system resources. It reduces the attack surface by minimizing the number of potential entry points for attackers. |
| 1. Adhere to the Principle of Least Privilege | Grant the minimum level of access necessary for users and processes to perform their functions. This principle limits the damage that can be done by a compromised account or process. By restricting access, the potential impact of a security breach is minimized. |
| 1. Sanitize Data Sent to Other Systems | Cleanse data before sending it to other systems to prevent the propagation of malicious content. This principle helps protect against injection attacks and data corruption by ensuring that data shared between systems is safe and well-formed. |
| 1. Practice Defense in Depth | Employ multiple layers of security controls to protect the system. Defense in depth ensures that if one security measure fails, others are in place to continue protecting the system. This approach increases the overall security posture by providing redundancies. |
| 1. Use Effective Quality Assurance Techniques | Implement rigorous quality assurance practices to identify and address security vulnerabilities. This includes code reviews, static analysis, and dynamic testing. Quality assurance helps ensure that the system is robust and secure before deployment. |
| 1. Adopt a Secure Coding Standard | Follow a consistent set of secure coding practices to reduce the risk of vulnerabilities. Secure coding standards provide guidelines for writing safe and secure code, helping developers avoid common pitfalls and ensure the reliability of the system. Adhering to these standards promotes best practices and enhances overall code quality. |

### 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] | Use Explicit Data Types: Using explicit data types helps ensure clarity and prevent unexpected behavior, as implicit conversions can lead to data corruption and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Assigning a character to an integer can lead to implicit conversion issues. |
| // Noncompliant  int x = 'a'; |

| **Compliant Code** |
| --- |
| Using the appropriate data type (char) for a character value ensures data integrity. |
| // Compliant  char x = 'a'; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Implicit data types can lead to subtle bugs and security vulnerabilities, but they are generally easy to detect and fix, especially with modern IDEs and compilers.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp – Avoid implicit conversion | This rule checks for instances where implicit type conversion may lead to unexpected behavior or security 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 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DAV] | Validate Data Ranges: Validating data ranges prevents out-of-bounds errors and potential security vulnerabilities due to invalid data inputs. |

| **Noncompliant Code** |
| --- |
| Assigning an invalid value to a variable without validation. |
| // Noncompliant  int age = -5; |

| **Compliant Code** |
| --- |
| Validating the age variable ensures it falls within a reasonable range. |
| // Compliant  int age = getUserInput();  if (age < 0 || age > 120) {  // Handle invalid age  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Invalid data ranges can lead to serious security vulnerabilities, such as buffer overflows, which are commonly exploited. It is critical to validate all data inputs to prevent such issues.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.3 | RangeChecker | Detects potential out-of-bounds access by validating data ranges during code analysis. |
| [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-STR] | Null-Terminate Strings: Ensuring strings are null-terminated prevents buffer overflows and other security vulnerabilities associated with string manipulation. |

| **Noncompliant Code** |
| --- |
| Failing to explicitly null-terminate a string can cause undefined behavior. |
| // Noncompliant  char str[10] = "hello"; |

| **Compliant Code** |
| --- |
| Null terminating the string ensures it is correctly handled by string functions. |
| // Compliant  char str[10] = "hello";  str[5] = '\0'; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Failing to null-terminate strings can lead to buffer overflows and undefined behavior, which can be exploited. While such issues are high-risk, they are typically easier to fix with proper string handling practices.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | StringTerminatorCheck | Checks for missing null terminators in strings, preventing potential buffer overflows. |
| [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-SQL] | Use Parameterized Queries: Parameterized queries prevent SQL injection attacks by ensuring user input is treated as data rather than executable code. |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries can lead to SQL injection. |
| // Noncompliant  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| Using parameterized queries treats user input as data, preventing SQL injection. |
| // Compliant  std::string query = "SELECT \* FROM users WHERE username = ?";  stmt = conn.prepareStatement(query);  stmt.setString(1, username); |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: SQL injection is one of the most dangerous vulnerabilities. Using parameterized queries is essential to prevent it, making this a critical security measure.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp- Avoid SQL Injection | Detects SQL queries that are vulnerable to injection attacks. |
| [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-MEM] | Avoid Dangling Pointers: Avoiding dangling pointers prevents access to invalid memory locations, reducing the risk of crashes and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Accessing memory after it has been freed leads to undefined behavior. |
| // Noncompliant  int\* ptr = new int(5);  delete ptr;  \*ptr = 10; |

| **Compliant Code** |
| --- |
| Setting the pointer to nullptr after deletion prevents accidental access. |
| // Compliant  int\* ptr = new int(5);  delete ptr;  ptr = nullptr; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Dangling pointers can lead to undefined behavior, including potential security vulnerabilities such as accessing unauthorized memory. These can be challenging to detect and fix, increasing the remediation cost.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18 | Memcheck | Detects use of memory after it has been freed, helping to prevent dangling pointer issues. |
| [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-ASS] | Use Assertions to Check Invariants: Assertions help catch logical errors during development by ensuring that conditions assumed to be true are indeed true. |

| **Noncompliant Code** |
| --- |
| The function assumes ptr is not null without checking. |
| // Noncompliant  void foo(int\* ptr) {  \*ptr = 5;  } |

| **Compliant Code** |
| --- |
| Using an assertion ensures ptr is not null before dereferencing it. |
| // Compliant  void foo(int\* ptr) {  assert(ptr != nullptr);  \*ptr = 5;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Assertions help catch logical errors during development but do not generally affect production code. However, missing or incorrect assertions can allow bugs to go unnoticed, leading to security issues.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | NullDereferenceChecker | Detects potential null pointer dereferences by checking for null pointer usage before dereferencing. |
| [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-EXC] | Catch Specific Exceptions: Catching specific exceptions rather than general ones allows for more precise error handling and debugging. |

| **Noncompliant Code** |
| --- |
| Catching all exceptions makes it difficult to diagnose the cause of an error. |
| // Noncompliant  try {  // Code that might throw  } catch (...) {  // Handle error  } |

| **Compliant Code** |
| --- |
| Catching specific exceptions allows for targeted error handling and easier debugging. |
| // Compliant  try {  // Code that might throw  } catch (const std::exception& e) {  // Handle specific error  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Catching all exceptions can obscure the root cause of errors, making debugging and security analysis more difficult. Specific exception handling is generally easy to implement, but it is crucial for robust and secure error handling.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp- Catch specific exceptions instead of generic ones | Detects code that catches all exceptions generically, encouraging the handling of specific exceptions for better 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** |
| --- | --- | --- |
| [Student Choice] | [STD-008-INP] | Validate All User Input: Validating all user input prevents malicious data from entering the system and causing harm. |

| **Noncompliant Code** |
| --- |
| Directly accepting user input without validation can introduce vulnerabilities. |
| // Noncompliant  std::cin >> userInput; |

| **Compliant Code** |
| --- |
| Validating user input ensures it meets expected criteria before processing. |
| // Compliant  std::cin >> userInput;  if (!isValid(userInput)) {  // Handle invalid input  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Unvalidated user input is a common vector for attacks such as SQL injection and cross-site scripting (XSS). Ensuring all input is validated is critical for security.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.10 | InputValidationChecker | Scans for unvalidated user inputs that could lead to security vulnerabilities such as injection attacks. |
| [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** |
| --- | --- | --- |
| [Student Choice] | [STD-009-LOG] | Log Security-Relevant Events: Logging security-relevant events helps in monitoring and responding to potential security incidents. |

| **Noncompliant Code** |
| --- |
| Not logging authentication attempts can hinder incident response. |
| // Noncompliant  void authenticate(User user) {  // Authenticate user  } |

| **Compliant Code** |
| --- |
| Logging authentication attempts aids in monitoring and detecting suspicious activity. |
| // Compliant  void authenticate(User user) {  log("Authentication attempt for user: " + user.name);  // Authenticate user  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Logging is essential for monitoring and responding to security incidents. While not immediately exploitable, insufficient logging can make it difficult to detect and respond to security breaches.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Splunk | 8.2 | EventLoggingChecker | Detects missing or incomplete logging of security-relevant events, helping to ensure that all critical actions are recorded. |
| [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** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CON] | Use Secure Default Configurations: Using secure default configurations minimizes the risk of misconfiguration and enhances the overall security posture. |

| **Noncompliant Code** |
| --- |
| Enabling insecure features by default can introduce vulnerabilities. |
| // Noncompliant  config.enableFeatureX = true; |

| **Compliant Code** |
| --- |
| Disabling insecure features by default ensures a more secure configuration. |
| // Compliant  config.enableFeatureX = false; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Explanation**: Insecure default configurations can leave a system vulnerable to attacks. Ensuring that all configurations are secure by default is critical for maintaining a strong security posture.

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Chef InSpec | 4.41 | DefaultConfigChecker | Scans configurations to ensure secure defaults are set, minimizing the risk of misconfigurations leading to security issues. |
| [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.

[Automation Summary for Green Pace Security Policy

Automation is a critical component in ensuring the enforcement and compliance of the coding standards defined in this security policy. Given Green Pace’s well-established DevOps process and infrastructure, the integration of automated tools into the DevSecOps pipeline will streamline the detection, prevention, and remediation of security vulnerabilities throughout the software development lifecycle.

Integration into the DevSecOps Pipeline:

Automation will be embedded at several key stages of the DevSecOps pipeline to provide continuous security checks and compliance enforcement:

1. Code Commit Stage (Pre-Commit Hooks)

- Tools Used: SonarQube, Cppcheck

- Purpose: As developers commit code to the version control system, pre-commit hooks will trigger static code analysis tools like SonarQube and Cppcheck. These tools will automatically scan the code for adherence to the defined coding standards, such as using explicit data types, validating input data ranges, and ensuring null-terminated strings. Any violations will be flagged immediately, preventing non-compliant code from entering the shared codebase.

2. Continuous Integration (CI) Pipeline:

- Tools Used: Jenkins, OWASP ZAP

- Purpose: During the CI process, the automation tools will perform both static and dynamic analysis. Jenkins will orchestrate the integration of tools like OWASP ZAP for dynamic security testing, particularly for standards related to SQL injection and input validation. This ensures that security vulnerabilities are identified early in the development process and before the code is merged into the main branch.

3. Continuous Monitoring and Compliance Checks:

- Tools Used: Splunk, Chef InSpec

- Purpose: post-deployment, continuous monitoring tools such as Splunk will track security-relevant events, ensuring that all actions are logged and analyzed for potential security incidents. Chef InSpec will be used to enforce secure default configurations and to regularly audit the system for compliance with the defined security standards. These tools will provide real-time alerts and reports, enabling prompt response to any deviations from the security policy.

4. Automated Remediation:

- Tools Used: Ansible, Puppet

- Purpose: In cases where non-compliant configurations or code are detected, automation tools like Ansible and Puppet can be configured to automatically remediate issues. This may include rolling back to a previous secure configuration, applying patches, or updating system settings to align with the security standards.

Modifications to Existing DevOps Process:

To effectively integrate these automation tools, the existing DevOps process will need to be modified as follows:

- Incorporate Pre-Commit Hooks: Modify the version control system to include pre-commit hooks for automated code scans.

- Extend CI/CD Pipelines: Expand the CI/CD pipelines to include stages for automated security testing and compliance checks.

- Integrate Continuous Monitoring Tools: Implement continuous monitoring tools across all environments to ensure ongoing compliance and quick identification of security breaches.

Conclusion:

By integrating these automation tools at strategic points in the DevSecOps pipeline, Green Pace will not only enforce the coding standards defined in this policy but also ensure continuous compliance, reduce manual effort, and enhance overall security posture. Automation will allow the development teams to focus on innovation while maintaining a strong, secure foundation for all software products.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-DAV | High | High | Medium | High | 5 |
| STD-003-STR | High | Medium | Medium | High | 4 |
| STD-004-SQL | High | High | Medium | High | 5 |
| STD-005-MEM | High | Medium | High | High | 4 |
| STD-006-ASS | Medium | Medium | Low | Medium | 3 |
| STD-007-EXC | Medium | Medium | Low | Medium | 3 |
| STD-008-INP | High | High | Medium | High | 5 |
| STD-009-LOG | Medium | Medium | Medium | Medium | 4 |
| STD-010-CON | High | Medium | Low | High | 4 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Create Policies for Encryption and Triple A

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

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

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

| **A. Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | * **What It Is: Encryption at rest refers to the protection of data stored on a disk or other persistent storage devices.** This includes databases, file systems, and any other storage media. * **How It Is Used**: Data at rest is encrypted using strong encryption algorithms, such as AES-256. This ensures that even if the storage media is physically compromised (e.g., stolen hard drive), the data remains protected and unreadable without the decryption key. * **Why and When the Policy Applies**: This policy applies to all sensitive data stored on any device within the organization. It is crucial for protecting customer information, intellectual property, and any other confidential data from unauthorized access. The policy ensures that encryption is consistently applied to all stored data, reducing the risk of data breaches. |
| Encryption in flight | * **What It Is**: Encryption in flight refers to the protection of data as it is transmitted over networks. This includes data being transferred between systems, services, or users. * **How It Is Used**: Data in transit is encrypted using protocols such as TLS (Transport Layer Security) to prevent interception and tampering by unauthorized parties. All network communications that involve sensitive data are secured with encryption. * **Why and When the Policy Applies**: This policy applies to all data transmissions involving sensitive or confidential information. By encrypting data in flight, the organization ensures that any data intercepted during transmission cannot be easily read or manipulated, maintaining the integrity and confidentiality of the information. |
| Encryption in use | * **What It Is**: Encryption in use refers to the protection of data while it is being processed or actively used in memory. This is the most challenging form of encryption but is important in highly sensitive environments. * **How It Is Used**: Data in use is encrypted in memory using techniques such as homomorphic encryption or secure enclave technology. This ensures that even if the system is compromised, the data remains protected. * **Why and When the Policy Applies**: This policy applies primarily in environments where highly sensitive data is processed and where there is a risk of memory-based attacks. While not all applications require encryption in use, it is essential for those handling critical data where maximum security is required. |

| **A. 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, device, or service before granting access to a system or resource. * **How It Is Used**: Authentication is implemented through multi-factor authentication (MFA), requiring users to provide at least two forms of verification (e.g., password and a one-time code sent to a mobile device) before accessing systems. * **Why and When the Policy Applies**: This policy applies to all systems and applications that handle sensitive data or provide access to critical resources. It ensures that only authorized individuals can access systems, significantly reducing the risk of unauthorized access and potential breaches. |
| Authorization | * **What It Is**: Authorization is the process of granting or denying access to resources based on a user’s role, permissions, and other contextual factors. * **How It Is Used**: Authorization is enforced through role-based access control (RBAC), where permissions are granted based on a user’s role within the organization. This limits access to resources only to those who need it to perform their duties. * **Why and When the Policy Applies**: This policy applies to all systems and applications within the organization. By strictly controlling who can access what resources, the policy minimizes the risk of data exposure and ensures that sensitive information is only accessible by those with the appropriate clearance. |
| Accounting | * **What It Is**: Accounting, often referred to as auditing, is the process of tracking user activities and access to resources within a system. It involves logging and analyzing access attempts, changes made, and any other relevant actions. * **How It Is Used**: All access to sensitive data, changes to critical systems, and other significant events are logged in detail. Regular audits are performed to ensure compliance with security policies and to detect any unauthorized or suspicious activity. * **Why and When the Policy Applies**: This policy applies to all systems handling sensitive data and critical resources. It ensures that there is a clear and traceable record of all actions within the system, which is essential for investigating security incidents, ensuring compliance with regulations, and maintaining overall system integrity. |

**\***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 standard and justify 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 best practice.

**[Coding Standard 1: Use Explicit Data Types**

**- Relevant Principles:**

**- Principle 9: Adopt a Secure Coding Standard**

**- Principle 2: Heed Compiler Warnings**

**- Justification:**

**- Adopt a Secure Coding Standard: This principle emphasizes the importance of following secure coding practices to avoid vulnerabilities. Using explicit data types reduces the likelihood of implicit conversions, which can introduce unexpected behavior and security issues, thus adhering to a secure coding standard.**

**- Heed Compiler Warnings: Explicit data types help in reducing compiler warnings related to type conversions. By using explicit types, developers can more easily identify and address potential issues flagged by the compiler.**

**Coding Standard 2: Validate Data Ranges**

**- Relevant Principles:**

**- Principle 1: Validate Input Data**

**- Principle 8: Practice Defense in Depth**

**- Justification:**

**- Validate Input Data: Validating data ranges is a direct application of this principle. By ensuring that input data falls within acceptable ranges, the risk of out-of-bound errors and other security vulnerabilities is minimized.**

**- Practice Defense in Depth: Validating data at multiple layers (e.g., user input, database input) provides additional layers of security, aligning with the defense in depth strategy.**

**Coding Standard 3: Null-Terminate Strings**

**- Relevant Principles:**

**- Principle 4: Keep It Simple**

**- Principle 9: Adopt a Secure Coding Standard**

**- Justification:**

**- Keep It Simple: Ensuring strings are null-terminated is a straightforward and effective way to prevent buffer overflows and other string-related vulnerabilities. Simplicity in string handling reduces the potential for errors.**

**- Adopt a Secure Coding Standard: This principle supports the use of established coding practices that prevent common vulnerabilities, such as buffer overflows, which are mitigated by proper string termination.**

**Coding Standard 4: Use Parameterized Queries**

**- Relevant Principles:**

**- Principle 7: Sanitize Data Sent to Other Systems**

**- \*\*Principle 8: Practice Defense in Depth**

**- Justification:**

**- Sanitize Data Sent to Other Systems: Parameterized queries ensure that user inputs are treated as data rather than executable code, effectively sanitizing inputs and preventing SQL injection attacks.**

**- Practice Defense in Depth: Implementing parameterized queries adds an additional layer of security by protecting the database from injection attacks, complementing other security measures.**

**Coding Standard 5: Avoid Dangling Pointers**

**- Relevant Principles:**

**- Principle 8: Practice Defense in Depth**

**- Principle 9: Adopt a Secure Coding Standard**

**- Justification:**

**- Practice Defense in Depth: Avoiding dangling pointers is critical in maintaining system stability and preventing memory-related vulnerabilities, which adds another layer of defense against potential exploits.**

**- Adopt a Secure Coding Standard\*\*: This principle encourages the following practices that prevent common issues like dangling pointers, which can lead to security vulnerabilities and system crashes.**

**Coding Standard 6: Use Assertions to Check Invariants**

**- Relevant Principles:**

**- Principle 6: Use Effective Quality Assurance Techniques**

**- Principle 4: Keep It Simple**

**- Justification:**

**- Use Effective Quality Assurance Techniques: Assertions are a key tool in verifying that assumptions about the code’s behavior are correct, helping to catch logical errors early in the development process.**

**- Keep It Simple: Assertions simplify debugging by immediately flagging violations of expected conditions, making it easier to identify and correct errors.**

**Coding Standard 7: Catch Specific Exceptions**

**- Relevant Principles:**

**- Principle 3: Architect and Design for Security Policies**

**- Principle 2: Heed Compiler Warnings**

**- Justification:**

**- Architect and Design for Security Policies: Catching specific exceptions aligns with designing for security by ensuring that errors are handled appropriately, reducing the risk of security issues arising from unhandled or improperly handled exceptions.**

**- Heed Compiler Warnings: By catching specific exceptions, developers can avoid broad, catch-all exception handling, which can lead to overlooked issues that might otherwise be flagged by the compiler.**

**Coding Standard 8: Validate All User Input**

**- Relevant Principles:**

**- Principle 1: Validate Input Data**

**- Principle 8: Practice Defense in Depth**

**- Justification:**

**- Validate Input Data: This standard directly applies to this principle by ensuring that all user input is validated to prevent malicious data from entering the system.**

**- Practice Defense in Depth: Validating input at multiple points in the application ensures that even if one validation layer is bypassed, others are in place to protect the system.**

**Coding Standard 9: Log Security-Relevant Events**

**- Relevant Principles:**

**- Principle 8: Practice Defense in Depth**

**- Principle 6: Use Effective Quality Assurance Techniques**

**- Justification:**

**- Practice Defense in Depth: Logging security-relevant events is crucial for maintaining an audit trail and detecting potential security breaches, providing an additional layer of defense by monitoring and responding to suspicious activities.**

**- Use Effective Quality Assurance Techniques: Logging helps in maintaining system reliability and security by ensuring that all actions are traceable, facilitating the identification of issues during audits and quality assurance processes.**

**Coding Standard 10: Use Secure Default Configurations**

**- Relevant Principles:**

**- Principle 5: Default Deny**

**- Principle 9: Adopt a Secure Coding Standard**

**- Justification:**

**- Default Deny: This principle supports the idea that systems should be secure by default, with all unnecessary features disabled. Secure default configurations ensure that the system is protected against common misconfigurations that could lead to security vulnerabilities.**

**- Adopt a Secure Coding Standard: Ensuring that secure defaults are used aligns with adopting secure practices that prevent vulnerabilities from arising due to improper configurations.]**

**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 comply with this policy.

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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [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 |