**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 data input into the system is validated to prevent malicious input from compromising the integrity or security of the software. This includes input validation techniques such as range checks, format checks, and data type validation. |
| 1. Heed Compiler Warnings | Pay close attention to compiler warnings and address them promptly. Compiler warnings often indicate potential vulnerabilities or errors in the code that could be exploited by attackers if left unresolved. |
| 1. Architect and Design for Security Policies | Incorporate security policies into the architecture and design of the software from the outset. This includes considering security requirements, threat modeling, and implementing security controls at every layer of the application. |
| 1. Keep It Simple | Favor simplicity in software design and implementation to reduce the attack surface and make the system easier to understand, maintain, and secure. Complex code and architectures are more prone to vulnerabilities and difficult to secure effectively. |
| 1. Default Deny | Adopt a default deny approach to access control, where all access is denied by default, and only explicitly authorized actions are allowed. This helps prevent unauthorized access and reduces the risk of security breaches. |
| 1. Adhere to the Principle of Least Privilege | Grant users and processes only the minimum level of access or permissions necessary to perform their tasks. By limiting access rights, the potential impact of security incidents or breaches can be minimized. |
| 1. Sanitize Data Sent to Other Systems | Ensure that data sent to external systems or components is properly sanitized to prevent injection attacks, such as SQL injection or cross-site scripting (XSS). Sanitization involves validating, escaping, or encoding data to neutralize potential threats. |
| 1. Practice Defense in Depth | Implement multiple layers of defense mechanisms and security controls throughout the software stack. This approach ensures that even if one layer is breached, other layers remain intact to mitigate the impact of security incidents. |
| 1. Use Effective Quality Assurance Techniques | Employ rigorous quality assurance (QA) techniques, including code reviews, static analysis, and automated testing, to identify and remediate security vulnerabilities early in the development lifecycle. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards, such as the SEI CERT C++ Coding Standard, to guide developers in writing secure code and to maintain consistency across the organization's software projects. A secure coding standard defines rules and best practices for writing code that is resilient to security threats. |

### 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** | **Validate Data Types** |
| --- | --- | --- |
| **Data Type** | [STD-001-DTP] | This standard ensures that all data types used in the code are appropriate and validated to prevent type-related vulnerabilities, such as data corruption or buffer overflows. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Using an incorrect data type without validation |
| int size = -1; |

| **Compliant Code** |
| --- |
| // Compliant description: Using the correct data type with validation |
| size\_t size = 10; |

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

| **Principles(s):** This standard aligns with the principle of "Validate Input Data" by ensuring that data types are validated to prevent vulnerabilities arising from incorrect or malicious input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.3 | TypeMismatch | Cppcheck is a static analysis tool for C/C++ code that can detect type mismatches and provide recommendations for correcting them. |
| Clang-Tidy | 13.0.0 | misc-misplaced-widening-cast | Clang-Tidy is a clang-based C++ linter tool that provides various checks for improving code quality, including checks for type mismatches. |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Sanitize Data Inputs** |
| --- | --- | --- |
| **Data Value** | [STD-002-DVL] | This standard ensures that all data inputs are sanitized to prevent injection attacks and ensure data integrity. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Using unsanitized user input directly |
| std::string username = GetUserInput(); |

| **Compliant Code** |
| --- |
| // Compliant description: Sanitizing user input before usage |
| std::string username = Sanitize(GetUserInput()); |

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

| **Principles(s):** This standard aligns with the principle of "Sanitize Data Sent to Other Systems" by ensuring that all data inputs are properly sanitized before usage, reducing the risk of injection attacks and maintaining data integrity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.2 | S3649 | SonarQube is a static code analysis tool that can detect security vulnerabilities, including unsanitized data inputs. Rule S3649 specifically focuses on detecting potentially dangerous uses of untrusted data. |
| Fortify Static Code Analyzer | 20.2 | Injection | Fortify Static Code Analyzer is a static analysis tool that specializes in identifying security vulnerabilities. The Injection checker specifically detects potential injection vulnerabilities in the codebase. |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Ensure String Termination** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | This standard ensures that all strings are properly null-terminated to prevent buffer overflows and string-related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Incorrect string termination |
| char buffer[10]; |

| **Compliant Code** |
| --- |
| // Compliant description: Proper string termination |
| char buffer[10] = "Hello"; |

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

| **Principles(s):** This standard aligns with the principle of "Validate Input Data" by ensuring that all strings are properly null-terminated, which helps prevent buffer overflows and related vulnerabilities. Additionally, it supports the principle of "Sanitize Data Sent to Other Systems" by ensuring that strings passed to other systems are correctly terminated, reducing the risk of injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2020.12 | STRING\_NULL\_TERM | Coverity is a static code analysis tool that can detect a wide range of software defects, including issues related to string termination. The STRING\_NULL\_TERM checker specifically identifies instances where strings may not be properly null-terminated, helping to prevent buffer overflows and related vulnerabilities. |
| Checkmarx | 9.3 | STR02-C | Checkmarx is a static application security testing (SAST) tool that identifies security vulnerabilities in source code. The STR02-C checker focuses on ensuring that strings are correctly null-terminated, helping to prevent buffer overflows and related vulnerabilities. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use Parameterized Queries** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | This standard ensures that SQL queries are parameterized to prevent SQL injection attacks and maintain data security. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Using string concatenation for SQL query |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| // Compliant description: Using parameterized query to prevent SQL injection |
| std::string query = "SELECT \* FROM users WHERE username = ?"; |

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

| **Principles(s):** This standard aligns with the principle of "Sanitize Data Inputs" by ensuring that SQL queries are parameterized, thus preventing SQL injection attacks. It also supports the principle of "Authenticate Everything" by maintaining data security and integrity through proper query construction. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S3649 | SonarQube is a static code analysis tool that detects security vulnerabilities in code. The S3649 checker specifically identifies instances where SQL queries are not parameterized, helping to prevent SQL injection attacks. |
| Fortify Static Code Analyzer | 21.1 | SQL\_INJECTION | Fortify Static Code Analyzer is a static application security testing (SAST) tool that identifies security vulnerabilities in source code. The SQL\_INJECTION checker focuses on detecting potential SQL injection vulnerabilities, including cases where SQL queries are not parameterized. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Implement Bounds Checking** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | This standard ensures that memory accesses are properly bound-checked to prevent buffer overflows and memory-related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Accessing memory without bounds checking |
| int array[10];  int value = array[15]; |

| **Compliant Code** |
| --- |
| // Compliant description: Bounds checking before accessing memory |
| if (index >= 0 && index < 10) {  int value = array[index];  } |

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

| **Principles(s):** This standard aligns with the principle of "Validate All Inputs" by ensuring that memory accesses are properly bound-checked before being accessed, thereby preventing buffer overflows and other memory-related vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | Bounds | Clang Static Analyzer is a static code analysis tool that detects bugs in C, C++, and Objective-C programs. The Bounds checker focuses on identifying out-of-bounds memory accesses, aiding in the prevention of buffer overflows and related vulnerabilities. |
| Coverity Static Analysis | 2021.12 | BUFFER\_SIZE | Coverity Static Analysis is a static application security testing (SAST) tool that identifies security vulnerabilities in source code. The BUFFER\_SIZE checker specifically detects instances where memory accesses are not properly bound-checked, helping to prevent buffer overflows and memory-related vulnerabilities. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions for Error Checking** |
| --- | --- | --- |
| **Assertions** | [STD-006-AST] | This standard ensures that assertions are used for error checking to detect and handle exceptional conditions effectively. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Lack of error checking using assertions |
| int result = Divide(a, b); |

| **Compliant Code** |
| --- |
| // Compliant description: Error checking using assertions |
| assert(b != 0);  int result = Divide(a, b); |

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

| **Principles(s):** This standard aligns with the principle of "Fail Fast" by ensuring that exceptional conditions are detected and handled effectively using assertions. Assertions provide a mechanism for early detection of errors, allowing for immediate termination of execution when exceptional conditions are encountered. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | AssertionFailure | Clang Static Analyzer detects assertion failures in C and C++ code. The AssertionFailure checker identifies instances where assertions are violated, allowing developers to detect and fix errors related to the absence or misuse of assertions for error checking. |
| GCC (GNU Compiler Collection) | 11.2.0 | -Werror | GCC is a compiler system that supports various programming languages, including C and C++. The -Werror flag enables the generation of compilation errors for failing assertions, ensuring that assertions are used for error checking during the build process. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Properly Handle Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | This standard ensures that exceptions are properly handled to prevent unexpected program termination and maintain system stability. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Lack of exception handling |
| void OpenFile(std::string filename) {  std::ifstream file(filename);  } |

| **Compliant Code** |
| --- |
| // Compliant description: Exception handling to handle file opening errors |
| void OpenFile(std::string filename) {  try {  std::ifstream file(filename);  } catch (const std::exception& e) {  }  } |

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

| **Principles(s):** This standard aligns with the principle of "Robustness" by ensuring that exceptions are properly handled to maintain system stability. Handling exceptions prevents unexpected program termination and enables graceful error recovery, contributing to the overall robustness of the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | ExceptionEscape | Clang Static Analyzer detects instances where exceptions are not properly handled in C and C++ code. The ExceptionEscape checker identifies code paths where exceptions can escape without being caught, leading to potential unexpected program termination. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Manage Memory Allocation** |
| --- | --- | --- |
| **Dynamic Memory Allocation** | [STD-008-DMA] | This standard ensures that dynamic memory allocation and deallocation are managed properly to prevent memory leaks and memory-related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Memory leak due to improper deallocation |
| int\* ptr = new int; |

| **Compliant Code** |
| --- |
| // Compliant description: Proper memory deallocation |
| int\* ptr = new int;  delete ptr; |

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

| **Principles(s):** This standard aligns with the principle of "Resource Management" by ensuring that dynamic memory allocation is managed properly. Proper memory allocation and deallocation prevent memory leaks, which could lead to system instability and resource exhaustion. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind's Memcheck tool is used to detect memory management errors, including memory leaks, in C and C++ programs. It provides detailed information about memory allocation and deallocation, helping to identify improper memory management practices such as memory leaks. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Implement Error Handling** |
| --- | --- | --- |
| **Error Handling** | [STD-009-ERR] | This standard ensures that error handling mechanisms are implemented to handle unexpected conditions and maintain system stability. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Lack of error handling |
| void ProcessData(std::string data) {  } |

| **Compliant Code** |
| --- |
| // Compliant description: Error handling to handle unexpected conditions |
| void ProcessData(std::string data) {  try {  } catch (const std::exception& e) {  }  } |

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

| **Principles(s):** This standard aligns with the principle of "Robustness" by ensuring that error handling mechanisms are in place to handle unexpected conditions gracefully. Proper error handling helps maintain system stability by preventing unexpected program termination and providing appropriate feedback to users. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.03 | Error Handling | Coverity is a static analysis tool that helps identify defects and vulnerabilities in source code. The Error Handling checker in Coverity detects instances where error handling mechanisms are missing or inadequate. It provides insights into potential areas where error handling should be implemented or improved. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use Strong Cryptography Algorithms** |
| --- | --- | --- |
| **Cryptography** | [STD-010-CRP | This standard ensures that strong cryptography algorithms are used to protect sensitive data and prevent unauthorized access. |

| **Noncompliant Code** |
| --- |
| // Noncompliant description: Using weak cryptography algorithm |
| std::string encrypted = WeakEncrypt(data); |

| **Compliant Code** |
| --- |
| // Compliant description: Using strong cryptography algorithm |
| std::string encrypted = StrongEncrypt(data); |

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

| **Principles(s):** This standard aligns with the principle of "Confidentiality" by ensuring that strong encryption algorithms are employed to safeguard sensitive data. By utilizing robust cryptographic techniques, the confidentiality of data is preserved, reducing the risk of unauthorized access and data breaches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA (Static Code Analyzer) | 20.2 | Cryptographic Vulnerabilities | Fortify SCA is a static code analysis tool that scans source code for security vulnerabilities, including cryptographic weaknesses. The Cryptographic Vulnerabilities checker in Fortify SCA identifies instances where weak cryptography algorithms are used, providing recommendations for using stronger alternatives. |

### 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 crucial role in enforcing and ensuring compliance with the standards outlined in this policy. Green Pace benefits from a robust DevOps process and infrastructure, depicted by the infinity symbol with "pre-production" on the left side and "production" on the right side. This symbolizes the continuous cycle of development, testing, and deployment.
* To integrate automation for policy enforcement, we can leverage existing DevOps practices. In the pre-production phase, where development and testing occur, automation tools can be introduced to scan code repositories for compliance with coding standards. Continuous integration (CI) pipelines can include static code analysis tools that automatically check for adherence to coding standards during code commits.
* As code progresses to the production phase, automated processes can monitor deployed applications for security vulnerabilities and policy violations in real-time. This includes implementing runtime application self-protection (RASP) solutions that dynamically analyze application behavior and detect anomalies or unauthorized access attempts.
* Furthermore, automation can extend to the configuration management and infrastructure provisioning stages. Infrastructure as code (IaC) tools such as Terraform or Ansible can enforce security configurations, ensuring that cloud environments and server configurations adhere to predefined security policies automatically.
* In summary, by integrating automation into the existing DevOps pipeline, Green Pace can effectively enforce and maintain compliance with the security standards outlined in this policy throughout the software development lifecycle.

### 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-DVL | High | Likely | High | High | 3 |
| STD-003-STR | Medium | Likely | Low | Medium | 2 |
| STD-004-SQL | High | Likely | High | High | 3 |
| STD-005-MEM | High | Likely | High | High | 3 |
| STD-006-AST | Medium | Likely | Medium | Medium | 2 |
| STD-007-EXC | Medium | Likely | Medium | Medium | 2 |
| STD-008-DMA | High | Likely | High | High | 3 |
| STD-009-ERR | Medium | Likely | Medium | Medium | 2 |
| STD-010-CRP | High | Likely | High | High | 3 |

### 

### 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 | This policy mandates the encryption of data when it is stored in databases or files. It applies to all data at rest within Green Pace's systems, ensuring that sensitive information remains protected even if unauthorized access occurs. |
| Encryption in flight | This policy requires the encryption of data during transmission over networks. It applies to all data transferred between systems or users within Green Pace's infrastructure, safeguarding it from interception or tampering. |
| Encryption in use | This policy governs the encryption of data while it is actively being processed or used within applications or systems. It ensures that sensitive information remains encrypted throughout its lifecycle, protecting it from unauthorized access or exposure. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This policy outlines the procedures and mechanisms for verifying the identities of users accessing Green Pace's systems. It covers user logins, ensuring that only authenticated individuals can gain access to the company's resources. |
| Authorization | This policy defines the rules and processes for granting access permissions to users based on their roles and responsibilities. It addresses the addition of new users and user level of access, ensuring that only authorized individuals have appropriate access to resources and databases. |
| Accounting | This policy governs the tracking and auditing of user activities within Green Pace's systems. It encompasses monitoring changes to the database and files accessed by users, providing accountability and traceability for all system interactions.  Top of Form  Top of Form |

**\***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

--------------------------------------------------------------------------------------------------------------------------------------------------

Connection:

1. **STD-001-CPP (Use Secure Coding Practices)**:
   * Principles: 1 (Principle of Least Privilege), 3 (Fail-Safe Defaults), 4 (Economy of Mechanism)
   * Justification:
     + Principle of Least Privilege (1): By following secure coding practices, developers ensure that code operates with minimal privileges, reducing the potential attack surface and limiting the impact of security breaches.
     + Fail-Safe Defaults (3): Secure coding practices emphasize defaulting to safe behaviors, such as sanitizing inputs and enforcing proper data validation, to prevent common vulnerabilities like injection attacks.
     + Economy of Mechanism (4): Secure coding practices advocate for simplicity and efficiency in code design, reducing the complexity of the system and making it easier to identify and mitigate security flaws.
2. **STD-002-DVL (Sanitize Data Inputs)**:
   * Principles: 3 (Fail-Safe Defaults), 7 (Least Astonishment)
   * Justification:
     + Fail-Safe Defaults (3): Sanitizing data inputs ensures that the system operates with safe defaults, preventing unexpected behavior or vulnerabilities resulting from malicious input.
     + Least Astonishment (7): Properly sanitized data inputs align with users' expectations, minimizing the risk of unexpected behavior or security vulnerabilities that could surprise users.
3. **STD-003-STR (Ensure String Termination)**:
   * Principles: 3 (Fail-Safe Defaults), 7 (Least Astonishment)
   * Justification:
     + Fail-Safe Defaults (3): Ensuring proper string termination by default prevents buffer overflows and related vulnerabilities, aligning with the principle of designing for safety by default.
     + Least Astonishment (7): Proper string termination practices ensure that software behaves predictably, avoiding unexpected outcomes or security issues that might astonish users.
4. **STD-004-SQL (Use Parameterized Queries)**:
   * Principles: 1 (Principle of Least Privilege), 4 (Economy of Mechanism)
   * Justification:
     + Principle of Least Privilege (1): Parameterized queries limit the privileges granted to SQL statements, reducing the risk of SQL injection attacks by enforcing separation of data and commands.
     + Economy of Mechanism (4): Parameterized queries simplify SQL statement construction, promoting efficiency and reducing the likelihood of errors or vulnerabilities associated with manual string concatenation.
5. **STD-005-MEM (Implement Bounds Checking)**:
   * Principles: 3 (Fail-Safe Defaults), 4 (Economy of Mechanism)
   * Justification:
     + Fail-Safe Defaults (3): Bounds checking by default ensures that memory accesses remain within safe limits, preventing buffer overflows and related vulnerabilities.
     + Economy of Mechanism (4): Bounds checking mechanisms simplify memory management, promoting efficiency and reducing the complexity of the system while enhancing security.
6. **STD-006-AST (Use Assertions for Error Checking)**:
   * Principles: 2 (Defense in Depth), 5 (Complete Mediation)
   * Justification:
     + Defense in Depth (2): Assertions provide an additional layer of defense by validating assumptions and checking for errors at runtime, complementing other error-handling mechanisms.
     + Complete Mediation (5): Assertions ensure complete mediation by checking for errors comprehensively, reducing the likelihood of unchecked exceptional conditions that could compromise system integrity.
7. **STD-007-EXC (Properly Handle Exceptions)**:
   * Principles: 5 (Complete Mediation), 7 (Least Astonishment)
   * Justification:
     + Complete Mediation (5): Proper exception handling ensures that all exceptional conditions are mediated and addressed, preventing unhandled errors that could lead to system instability.
     + Least Astonishment (7): Effective exception handling aligns with users' expectations of system behavior, minimizing surprises caused by unanticipated errors or failures.
8. **STD-008-DMA (Manage Memory Allocation)**:
   * Principles: 1 (Principle of Least Privilege), 3 (Fail-Safe Defaults)
   * Justification:
     + Principle of Least Privilege (1): Proper memory management reduces the risk of unauthorized access or exploitation by limiting the privileges associated with memory allocation and deallocation.
     + Fail-Safe Defaults (3): Effective memory allocation management establishes safe defaults, preventing memory-related vulnerabilities and unexpected behavior resulting from improper memory handling.
9. **STD-009-ERR (Implement Error Handling)**:
   * Principles: 5 (Complete Mediation), 6 (Least Common Mechanism)
   * Justification:
     + Complete Mediation (5): Robust error handling mechanisms ensure complete mediation of unexpected conditions, minimizing the impact of errors and preventing system instability.
     + Least Common Mechanism (6): Standardized error handling practices promote consistency and reduce complexity, enhancing system reliability and maintainability.
10. **STD-010-CRP (Use Strong Cryptography Algorithms)**:
    * Principles: 2 (Defense in Depth), 8 (Least Privilege)
    * Justification:
      + Defense in Depth (2): Strong cryptography algorithms add an additional layer of defense to protect sensitive data, complementing other security measures and mitigating the impact of potential breaches.
      + Least Privilege (8): By using strong cryptography algorithms, access to sensitive data is restricted to authorized entities only, aligning with the principle of limiting privileges to the minimum necessary level.

This mapping demonstrates how each standard is supported by fundamental security principles, ensuring that Green Pace's security policy is rooted in best practices and widely accepted principles.

--------------------------------------------------------------------------------------------------------------------------------------------------

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 | 04/01/2024 | Module Three Milestone | Laynie Tierney | Laynie Tierney |
| 2.0 | 04/17/2024 | Project One | Laynie Tierney | Laynie Tierney |
| 3.0 | 04/20/2024 | Project One (Update) | Laynie Tierney | Laynie Tierney |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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