**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 6](#_Toc52464060)

[Coding Standard 3 7](#_Toc52464061)

[Coding Standard 4 8](#_Toc52464062)

[Coding Standard 5 9](#_Toc52464063)

[Coding Standard 6 10](#_Toc52464064)

[Coding Standard 7 12](#_Toc52464065)

[Coding Standard 8 14](#_Toc52464066)

[Coding Standard 9 15](#_Toc52464067)

[Coding Standard 10 16](#_Toc52464068)

[Defense-in-Depth Illustration 18](#_Toc52464069)

[Project One 18](#_Toc52464070)

[1. Revise the C/C++ Standards 18](#_Toc52464071)

[2. Risk Assessment 18](#_Toc52464072)

[3. Automated Detection 18](#_Toc52464073)

[4. Automation 18](#_Toc52464074)

[5. Summary of Risk Assessments 19](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 20](#_Toc52464076)

[7. Map the Principles 21](#_Toc52464077)

[Audit Controls and Management 24](#_Toc52464078)

[Enforcement 24](#_Toc52464079)

[Exceptions Process 24](#_Toc52464080)

[Distribution 25](#_Toc52464081)

[Policy Change Control 25](#_Toc52464082)

[Policy Version History 25](#_Toc52464083)

[Appendix A Lookups 25](#_Toc52464084)

[Approved C/C++ Language Acronyms 25](#_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 data is checked for correctness to prevent malicious data from exploiting vulnerabilities using input validation, and sanitization to mitigate SQL injection and buffer overflows. |
| 1. Heed Compiler Warnings | Usually indicate potential security issues like code vulnerabilities or logic errors. |
| 1. Architect and Design for Security Policies | Defining clear policies and ensuring the system design adheres to the security policy, with things like separation of duties and secure defaults. |
| 1. Keep It Simple | Complex systems are harder to understand and audit, by keeping it simple the likely possibility of security flaws is reduced and are easier to test and maintain. |
| 1. Default Deny | Configuring a system to deny access by default by denying all but those with permission. |
| 1. Adhere to the Principle of Least Privilege | Users should only have the minimum level of access to perform their duties, by doing this you prevent damage from accidental or malicious actions. |
| 1. Sanitize Data Sent to Other Systems | This is used to prevent injections and data corruption, by encoding output data, escaping characters, and making sure data is in the proper format. |
| 1. Practice Defense in Depth | Implementation of multiple layers of security controls, using a combination of different preventative, detective, and response measures so if one layer fails others can cover. |
| 1. Use Effective Quality Assurance Techniques | Using code reviews, automated testing, static analysis, and dynamic analysis can help identify and fix security vulnerabilities early and cause less problems later. |
| 1. Adopt a Secure Coding Standard | This ensures that developers follow best practices for writing secure code from the start and provide the guidelines for avoiding common errors and vulnerabilities. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Variable Names: Descriptive names improve code readability and maintainability to make it easy to understand the purpose of each variable. |

| **Noncompliant Code** |
| --- |
| Variable names are not descriptive and use ambiguous abbreviations. |
| int a;  float bal;  std::string un; |

| **Compliant Code** |
| --- |
| Variable names are descriptive and use camelCase. Member variables are prefixed with **m\_**. |
| int userAge;  float accountBalance;  std::string m\_userName; |

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

| **Principles(s):** Readability and maintainability which is descriptive variable names enhance code readability by clearly indicating the purpose and use of each variable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0 | Readability-identifier-naming | Popular tool for C++ code analysis and refactoring. Allows enforced naming conventions to ensure names are descriptive and follow styles. |
| SonarQube | 9.9 LTS | S1197 (naming convention) | Provides static code analysis. |
| Cppcheck | 2.14 | VariableNaming | Static analysis tool for C/C++ can be configured to verify that variable names are descriptive and adhere to naming conventions. |
| CodeQL | 1.13.1 | Cpp/naming-convention | Code analysis tool to detect various code quality issues. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Magic Numbers: use named constants in place of magic numbers to improve code readability and maintainability. |

| **Noncompliant Code** |
| --- |
| Using raw numbers in the code. |
| for (int i = 0; i < 5; ++i) {  ...  } |

| **Compliant Code** |
| --- |
| Use named constants instead of magic numbers. |
| const int MAX\_RETRIES = 5;  for (int i = 0; i < MAX\_RETRIES; ++i) {  ...  } |

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

| **Principles(s):** Readability and maintainability using named constants improves code readability by providing names of values that might be unclear. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Eslint | 9.5 | No-magic-numbers | Static code analysis tool can be adapted for C++ code using config and plugins. |
| Cppcheck | 2.14 | MISRA-C++:2008-5.2.1 | Encourages the use of symbolic constants instead of literals. |
| Clang-Tidy | 19.0 | cppcoreguidelines-avoid-magic-numbers | Helps enforce the use of named constant over MN. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Proper String Initialization and Termination: Crucial to preventing buffer overflows and memory corruption. |

| **Noncompliant Code** |
| --- |
| Using uninitialized or unterminated strings. |
| char str[10];  strncpy(str, "Hello", 5); |

| **Compliant Code** |
| --- |
| Initialize strings with valid values and ensure proper null termination. |
| char str[10] = {'\0'};  strncpy(str, "Hello", sizeof(str) - 1); |

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

| **Principles(s):** Robustness and security properly initializing and terminating strings prevents buffer overflows and memory corruption vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity Static Analysis | 2023.12 | UninitializedVariable | Can detect uninitialized variables and strings, to identify and fix issues. |
| PVS-Studio | 7.31 | V806 | Helps detect issues related to string initialization and termination. |
| Clang Analyzer | 18.1.6 | clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling | Can identify unsafe string handling practices including improper initialization and termination. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use Parameterized Queries or Prepared Statements: This prevents SQL attacks by separating SQL code from user input, while the queries bind the input parameters to placeholders in the query preventing malicious input. |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE username='" + username + "'"; |

| **Compliant Code** |
| --- |
| Use parameterized queries or prepared statements to safely handle user input. |
| std::string username = getUserInput();  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):** Security SQL injection prevention, separates queries of user input and parameterized or prepared statements prevents injections. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.15 | SQL Injection Scanner | Can automatically detect vulnerabilities by testing various input parameters and payloads. |
| Fortify Static Code Analyzer | 23.1 | SQL Injection (category) | Uses dedicated rules and analysis capabilities to identify SQL Inject. |
| Checkmarx | 9.0 | SQL Injection (category) | Comprehensive static analysis and flags instances where user input is not properly sanitized. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Validate Memory Operations: Helps prevent buffer overflows, memory leaks and other memory related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Performing memory operations without proper validation. No Bounds Checking. |
| char buffer[10];  std::strcpy(buffer, "Hello, world!"); |

| **Compliant Code** |
| --- |
| Validate memory operations to ensure they stay within the bounds of allocated memory. Bounds Checking. |
| char buffer[10];  if (std::strlen("Hello, world!") < sizeof(buffer)) {  std::strcpy(buffer, "Hello, world!");  } else {  // Handle error (buffer overflow)  } |

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

| **Principles(s):** Robustness and security. Validating memory operations ensures code does not attempt read or write outside of the bounds of allocated memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.23 | Memcheck | Detects memory management problems. |
| AddressSanitizer | Integrated | GCC 10.3.0 / Clang 12.0.1 | Runtime memory error detector. |
| Cppcheck | 2.14 | Buffer overflow | Rules to detect buffer overflows and improper memory operations. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Document Assumptions with Assertions: Helps improve code readability and maintainability and provides insight into the intended behavior of the code. |

| **Noncompliant Code** |
| --- |
| Using assertions without documenting the underlying assumptions. |
| void process(int value) {  assert(value >= 0);  // Process value  } |

| **Compliant Code** |
| --- |
| Document assumptions with assertions to enhance code readability. |
| void process(int value) {  // Ensure value is non-negative  assert(value >= 0);  // Process value  } |

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

| **Principles(s):** Readability and maintainability. Documenting assumptions and assertions provides clarity about how the code is expected to operate. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0 | misc-dangling-handle, readability-assertion-documentation | Checks for various coding practices. |
| SonarQube | 9.9 LTS | S3923 | Rule helps ensure assertions are used appropriately. |
| Cppcheck | 2.14 | assertWithSideEffects, AssertWithoutComment | Checks to make sure assertions are used correctly. Can also check comments explaining them. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catch Specific Exceptions: Allows for more precise error handling and recovery strategies by differentiating between different types of errors. |

| **Noncompliant Code** |
| --- |
| Catching exceptions without specifying the exception type. |
| void process(int value) {  try {  // Code that may throw exceptions  } catch (...) {  // Generic error handling  }  } |

| **Compliant Code** |
| --- |
| Catch specific exceptions to handle different error scenarios. |
| void process(int value) {  try {  // Code that may throw exceptions  } catch (const std::invalid\_argument& e) {  // Handle invalid argument error  } catch (const std::runtime\_error& e) {  // Handle runtime error  }  } |

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

| **Principles(s):** Precision in error handling. By catching specific exceptions code can implement precise error handling and recovery strategies tailored to different errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0 | misc-throw-by-value-catch-by-reference | Ensures specific exceptions are caught and handled. |
| SonarQube | 9.9 LTS | S2486 | Rule that helps catch exceptions specifically. |
| Cppcheck | 2.14 | ExceptionCatching, catchAll | Provides checks to detect generic exception handling and recommended catching specific exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| SQL Injection | [STD-008-CPP] | Sanitize Input Data: This involves validating and filtering user input to remove or escape characters that could be interpreted as SQL commands. |

| **Noncompliant Code** |
| --- |
| Not sanitizing user input before using it in SQL queries. |
| std::string userInput = getUserInput();  std::string query = "SELECT \* FROM users WHERE username='" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Sanitize user input to remove or escape characters that could be used in SQL injection attacks. |
| std::string userInput = getUserInput();  // Sanitize userInput before using it in the SQL query |

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

| **Principles(s):** Security SQL injection. Validating and sanitizing user input to catch special characters or malicious code embedded in user inputs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.15 | Input Validation Scanner | Can scan input fields and detect SQL injection by testing with various payloads. |
| Fortify Static Code Analyzer | 23.1 | SQL Injection (category) | Identifies SQL injection vulnerabilities by scanning for improper handling of user inputs. |
| SonarQube | 9.9 LTS | S3649 | Rule focusing on detecting and preventing SQL injections |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | [STD-009-CPP] | Use Smart Pointers: Helps prevent memory leaks and dangling pointers by automatically managing memory allocation and deallocation. |

| **Noncompliant Code** |
| --- |
| Manually managing memory allocation and deallocation with raw pointers. |
| int\* ptr = new int;  ...  delete ptr; |

| **Compliant Code** |
| --- |
| Use smart pointers to automatically manage memory allocation and deallocation. |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(); |

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

| **Principles(s):** Resource management and safety. Using smart pointers so memory is automatically and correctly managed preventing common issues like memory leaks and dangling pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0 | modernize-use-using, cppcoreguidelines-owning-memory | Includes checks to modernize code using smart pointers instead of raw. |
| SonarQube | 9.9 LTS | RSPEC-2259 | Rule that helps ensure smart pointers are used instead of raw. |
| Cppcheck | 2.14 | useSmartPointer | Detects the use of raw pointers and recommends smart pointers. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exceptions | [STD-010-CPP] | Ensure Exception Safety: Essential for maintaining program integrity when exceptions exist, by designing functions and data structures to guarantee consistency and prevent resource leaks. |

| **Noncompliant Code** |
| --- |
| Code that is not exception-safe and may lead to resource leaks or inconsistent program state. |
| void process() {  // Allocate resources  // Perform operations that may throw exceptions  // Release resources only if no exceptions occur  } |

| **Compliant Code** |
| --- |
| Design functions and data structures to ensure exception safety. |
| void process() {  // Allocate resources using RAII (Resource Acquisition Is Initialization)  ResourceHolder holder;  // Perform operations that may throw exceptions  // Resource deallocation handled automatically by RAII  } |

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

| **Principles(s):** Robustness and resource management. Exception safety guarantees a program remains in consistent states and resources are properly managed even when exceptions occur. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0 | cppcoreguidelines-no-malloc, cppcoreguidelines-owning-memory | Enforce exception safety practices like avoiding manual memory management and encourage use of RAII. |
| SonarQube | 9.9 LTS | RSPEC-2737 | Rule to help identify issues with exception safety. |
| Cppcheck | 2.14 | resourceLeak, exceptionSafety | Rules to detect resource leaks and ensure exception safety. |

### 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 with 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.

There are many ways to integrate automation enforcement of the standards that would allow for better optimization of current standards already in place. Starting with static and dynamic analysis tools that will allow for early detection of security vulnerabilities during the development phase, then continuing with continuous integration and deployment tools to ensure security checks are integrated into every stage of the development pipeline. It would also be good to implement security information and event management systems for real time monitoring and detection of security incidents and having automated response tools to have an immediate response to detected threats and minimize the down time.

By implementing these automation tools Green Pace can have continuous enforcement and compliance with security standards enhancing the overall security of their applications.

### 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-CPP | Medium | Medium | Low | Medium | 2 |
| STD-003-CPP | High | Medium | Medium | High | 5 |
| STD-004-CPP | Critical | Medium | Medium | High | 5 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 2 |
| STD-007-CPP | Medium | High | Medium | High | 3 |
| STD-008-CPP | Critical | High | Medium | High | 5 |
| STD-009-CPP | High | Medium | Medium | High | 4 |
| STD-010-CPP | High | Medium | Medium | High | 5 |
| [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 encryptions (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 | It refers to the protection of data stored on physical media which includes databases, file systems and other storage solutions where data is stored when not being actively accessed.  All sensitive data must be encrypted at rest using encryption algorithms like AES-256 and encryption keys mut be stored securely using hardware security modules or key management services to make sure they aren’t exposed.  The reason it should be used is to ensure even if physical security is breached the data remains inaccessible without the decryption keys, protecting against data theft and complying with data protection regulations. |
| Encryption in flight | This refers to securing data as it moves across networks and preventing interception and unauthorized access to data being transmitted.  All data transmitted over networks must be encrypted using secure protocols like transport layer security 1.2 or higher. This applies to data being transferred via the internet, internal networks, and between cloud services.  The reason it should be used it to protect against man in the middle attacks, eavesdropping, and data breaches during transmission, making sure data integrating and confidentiality are maintained. |
| Encryption in use | This refers to securing data while it is being processed and that the data remains protected during operations in memory.  Data being protected during this process is using technologies such as homomorphic encryption, secure enclaves, and trusted execution environments.  The reason it should be used is that it ensures data remains secure even when being processed reducing the risk of exposure from memory attacks or other forms of in-process data leaks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is the process of verifying the identity of a user or system. It makes sure only authorized users can access resources.  All systems should have strong authentication mechanisms like multi-factor authentication to verify user identities, and logs must be maintained and regularly reviewed for potential issues with users use.  Why strong authentication matters are it can prevent unauthorized access, protects credential theft, and allows only legitimate users so they can access sensitive systems and data. |
| Authorization | Determines what an authenticated user is allowed to do or have access to. It controls access levels and permission to resources.  These are important and must be implemented to ensure users have the minimum necessary access to perform their duties using principles of least privilege. Role-based access control or attribute-based access control models should be used as well.  Proper authorization makes it so users can only access data and resources necessary to perform their roles, minimizing the risk of data breaches and minimizing damage bad actors can do if they do gain access. |
| Accounting | This involves tracking user activities and maintaining logs for review and analysis. This ensures traceability of action and accountability of users’ actions.  All systems must log user activities like logins, changes to databases, access to files, and modifications to user permissions. The logs are securely stored and regularly reviewed for suspicious activity.  This provides visibility into user activity, enabling the detection and investigation of abnormal activities. It also follows compliances with regulatory requirements and facilitates forensic analysis in the event of an incident. |

**\***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 best practice.

Standard 1 Variable Names

Principles - 4, 9, 10

Keep it simple 4- using descriptive and consistent variable names makes code easier to read, understand and maintain reducing errors.

Use effective quality assurance techniques 9- Clear variable names facilitate code reviews and automated testing, helping identify potential issues early.

Adopt a secure coding standard 10- Consistent naming conventions are part of secure coding practices which prevent common errors and improve code quality.

Standard 2 Magic Numbers

Principles – 4, 9, 10

Keep It Simple 4- Replacing magic numbers with named constants makes the code more understandable and maintainable.

Use Effective Quality Assurance Techniques 9- Named constants improve readability, aiding in code reviews and testing.

Adopt a Secure Coding Standard 10- Avoiding magic numbers is a best practice in secure coding, ensuring values are meaningful and maintainable.

Standard 3 Proper String Initialization and Termination

Principles – 1, 7, 10

Validate Input Data 1- Proper initialization and termination of strings prevent buffer overflows, ensuring data integrity.

Sanitize Data Sent to Other Systems 7- Correctly handled strings avoid unintended data corruption and injection vulnerabilities.

Adopt a Secure Coding Standard 10- Proper string handling is crucial in secure coding practices to prevent common vulnerabilities.

Standard 4 Use Parameterized Queries or Prepared Statements

Principles – 1, 7, 8, 10

Validate Input Data 1- Parameterized queries prevent SQL injection by separating data from code.

Sanitize Data Sent to Other Systems 7- Properly parameterized queries ensure that user inputs are correctly handled before being sent to the database.

Practice Defense in Depth 8- Using parameterized queries adds a layer of security, protecting the database even if other defenses fail.

Adopt a Secure Coding Standard 10- Using parameterized queries is a best practice in secure coding to prevent SQL injection attacks.

Standard 5 Validate Memory Operations

Principles – 1, 9, 10

Validate Input Data 1- Validating memory operations ensures that data fits within allocated memory, preventing buffer overflows.

Use Effective Quality Assurance Techniques 9- Proper validation makes it easier to detect and fix memory-related issues during testing.

Adopt a Secure Coding Standard 10- Ensuring memory operations are validated is a crucial aspect of secure coding to prevent vulnerabilities.

Standard 6 Document Assumptions with Assertions

Principles – 4, 9, 10

Keep It Simple 4- Documenting assumptions with assertions makes the code more understandable and maintainable.

Use Effective Quality Assurance Techniques 9- Assertions help identify logic errors and incorrect assumptions early in the development process.

Adopt a Secure Coding Standard 10- Using assertions is part of secure coding practices to ensure the code behaves as expected.

Standard 7 Catch Specific Exceptions

Principles – 8, 9, 10

Practice Defense in Depth 8- Catching specific exceptions allows for precise error handling, adding a layer of robustness to the application.

Use Effective Quality Assurance Techniques 9- Specific exception handling improves the reliability of the code, making it easier to test and debug.

Adopt a Secure Coding Standard 10- Catching specific exceptions is a best practice in secure coding to handle different error scenarios appropriately.

Standard 8 Sanitize Input Data

Principles – 1, 7, 10

Validate Input Data 1- Sanitizing input data ensures that user inputs do not contain malicious data, preventing injections and other vulnerabilities.

Sanitize Data Sent to Other Systems 7- Proper sanitization prevents data corruption and ensures that data sent to other systems is safe.

Adopt a Secure Coding Standard 10- Sanitizing input data is a fundamental practice in secure coding to protect applications from malicious inputs.

Standard 9 Use Smart Pointers

Principles – 4, 9, 10

Keep It Simple 4- Smart pointers simplify memory management, making the code easier to understand and maintain.

Use Effective Quality Assurance Techniques 9- Smart pointers help detect and prevent memory leaks and dangling pointers during testing.

Adopt a Secure Coding Standard 10- Using smart pointers is a best practice in secure coding to ensure proper memory management.

Standard 10 Ensure Exception Safety

Principles – 3, 4, 8, 10

Architect and Design for Security Policies 3- Designing for exception safety ensures that the system adheres to security policies by maintaining consistent state and resource management.

Keep It Simple 4- Exception-safe code is easier to understand and maintain, reducing the likelihood of errors.

Practice Defense in Depth 8- Exception safety adds a layer of reliability, ensuring that resources are correctly managed even when exceptions occur.

Adopt a Secure Coding Standard 10- Ensuring exception safety is a critical practice in secure coding to maintain program integrity and prevent resource leaks.

**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 the 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 | 06/16/2024 | Project 1 additions | Ren Welck |  |
| [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 |