

# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating Input Data is crucial in safeguarding software applications. Developers can mitigate risks such as injection attacks and buffer overflows by rigorously inspecting and confirming all incoming data before processing. This proactive approach helps maintain data integrity and system reliability, reducing the likelihood of unauthorized access and malicious exploits. Effective input validation includes verifying data format, length, and type adherence to expected criteria, ensuring that only safe and valid inputs proceed for further processing. Organizations bolster their security defenses by implementing robust input validation practices without overly burdening users or compromising system performance. |
| 1. Heed Compiler Warnings | It's crucial to take compiler warnings seriously in software development. Ignoring these alerts can lead to potential vulnerabilities and bugs in the codebase. By addressing compiler warnings promptly, developers ensure a cleaner, more secure code that minimizes the risk of exploitable weaknesses. |
| 1. Architect and Design for Security Policies | It is essential to design software with security policies in mind from the outset. This approach ensures that security measures are integrated into the architecture and design phases rather than added as an afterthought. Developers create robust systems that adhere to industry standards and best practices by aligning with established security policies. |
| 1. Keep It Simple | Simplifying software design and implementation is critical to enhancing security. Complex systems tend to introduce more vulnerabilities and are more challenging to maintain and secure. By keeping solutions straightforward and focused, developers reduce the potential for errors and improve overall system resilience. |
| 1. Default Deny | Adopting a default deny approach means denying all access to resources by default and granting permissions only when explicitly authorized. This principle minimizes the attack surface by restricting unnecessary access and reduces the risk of unauthorized activities or breaches. |
| 1. Adhere to the Principle of Least Privilege | Following the principle of least privilege ensures that users and processes are granted only the minimum permissions necessary to perform their tasks. This restriction limits potential damage from compromised accounts or malicious activities, bolstering overall system security. |
| 1. Sanitize Data Sent to Other Systems | Validating and sanitizing data before transmission is critical for preventing injection attacks and ensuring data integrity. By scrubbing inputs for malicious content and validating data formats, developers mitigate unauthorized data manipulation or exploitation risks. |
| 1. Practice Defense in Depth | Implementing multiple layers of security defenses provides comprehensive protection against various threats. This strategy combines preventive, detective, and corrective controls across different levels of the system architecture to create a robust and resilient security posture. |
| 1. Use Effective Quality Assurance Techniques | Employing rigorous quality assurance practices helps identify and rectify security flaws and vulnerabilities early in the development lifecycle. Thorough testing, code reviews, and vulnerability assessments contribute to building more secure and reliable software products. |
| 1. Adopt a Secure Coding Standard | Establishing secure coding standards ensures consistency and adherence to best practices throughout development. Using standardized guidelines and practices, developers reduce the likelihood of introducing vulnerabilities and improve overall code quality and security posture. |

### 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 | Ensure correct and appropriate usage of data types to optimize memory usage and prevent unintended type conversions or overflows. |

| **Noncompliant Code** |
| --- |
| Assigning a floating-point value to an integer without explicit casting |
| int x = 5.5; |

| **Compliant Code** |
| --- |
| Assigning an integer value to an integer variable |
| int x = 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** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp | Ensures correct and appropriate usage of data types |
| PVS-Studio | 7.15 | V126 | Checks for incorrect type conversions and potential overflows |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Maintain consistent and valid values for data variables to ensure predictable behavior and prevent bugs due to uninitialized or invalid values. |

| **Noncompliant Code** |
| --- |
| Variable declared but not initialized |
| int count; |

| **Compliant Code** |
| --- |
| Initialize variable with a valid initial value |
| int count = 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** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp | Detects uninitialized variables |
| PVS-Studio | 7.15 | V104 | Checks for uninitialized or inconsistent data values |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Use proper string handling techniques to prevent buffer overflows, null termination issues, and ensure compatibility across different platforms. |

| **Noncompliant Code** |
| --- |
| Declaring a fixed-size array without ensuring null termination and then copying a string without checking buffer size |
| char str[9];  strcpy(str, "Hello, world!"); |

| **Compliant Code** |
| --- |
| Use std::string for dynamic size and automatic null termination |
| std::string str = "Hello, world!"; |

**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 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Ensures proper string handling to prevent buffer overflows |
| PVS-Studio | 7.15 | V512 | Detects incorrect string handling techniques |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL injection vulnerabilities by using parameterized queries and avoiding dynamic SQL construction with user input. |

| **Noncompliant Code** |
| --- |
| User input with malicious intent combined with dynamic SQL construction |
| std::string username = "admin'; DROP TABLE users; --";  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| Use parameterized queries with placeholders for user input |
| std::string username = "admin";  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):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Identifies potential SQL injection vulnerabilities |
| PVS-Studio | 7.15 | V1037 | Detects dynamic SQL queries with user input |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Manage memory effectively to avoid memory leaks, dangling pointers, and buffer overflows, ensuring robust and secure application behavior. |

| **Noncompliant Code** |
| --- |
| Missing deletion of allocated memory, leading to memory leak |
| int\* ptr = new int; |

| **Compliant Code** |
| --- |
| Use smart pointers for automatic memory management and avoid manual deallocation |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(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** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17 | Memcheck | Detects memory leaks and dangling pointers |
| PVS-Studio | 7.15 | V112 | Checks for potential memory leaks and improper memory management |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use assertions to validate assumptions and program state during development and debugging, improving code reliability and identifying issues early. |

| **Noncompliant Code** |
| --- |
| Using assert without a meaningful message |
| assert(x > 0); |

| **Compliant Code** |
| --- |
| Provide a meaningful message with assert |
| assert(x > 0 && "x must be greater than 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** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Ensures meaningful assertions are used |
| PVS-Studio | 7.15 | V595 | Detects improper or missing assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle exceptional conditions gracefully using C++ exception handling mechanisms, ensuring robust error recovery and program stability. |

| **Noncompliant Code** |
| --- |
| Exception is thrown but the catch block is empty |
| try {  throw SomeException("An error occurred");  } catch (...) {  // Empty catch block  } |

| **Compliant Code** |
| --- |
| Handling specific exception types |
| try {  throw SomeException("An error occurred");  } catch (const SomeException& ex) {  std::cerr << "SomeException caught: " << ex.what() << std::endl;  } catch (const std::exception& ex) {  std::cerr << "Exception caught: " << ex.what() << std::endl;  } catch (...) {  std::cerr << "Unknown exception caught" << std::endl;  } |

**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 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Ensures proper exception handling |
| PVS-Studio | 7.15 | V610 | Detects potential issues in exception handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Validate Input Data | STD-008-CPP | Validate and sanitize input data to prevent vulnerabilities such as buffer overflows, injection attacks, and data corruption, ensuring robustness and security of the application. |

| **Noncompliant Code** |
| --- |
| Process user input without validating length or content |
| void processUserInput(char\* userInput) {  strcpy(buffer, userInput);  } |

| **Compliant Code** |
| --- |
| Input validation to prevent buffer overflow |
| void processUserInput(const char\* userInput) {  if (strlen(userInput) < BUFFER\_SIZE) {  strcpy(buffer, userInput);  } else {  }  } |

**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** |
| --- | --- | --- | --- | --- |
| Critical | High | Medium | Critical | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Validates and sanitizes input data |
| PVS-Studio | 7.15 | V106 | Detects unsanitized user input |

**Coding Standard 9**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Keep It Simple | STD-009-CPP | Simplify code implementation to enhance readability, maintainability, and reduce the risk of introducing bugs or vulnerabilities due to complexity. |

| **Noncompliant Code** |
| --- |
| Overly complex function |
| double calculateInterest(double principal, double rate, int years) {  double result = principal \* pow((1 + rate / 100), years);  return result;  } |

| **Compliant Code** |
| --- |
| Simplified function for interest calculation |
| double calculateInterest(double principal, double rate, int years) {  double interest = rate / 100;  double amount = principal \* pow((1 + interest), years);  return amount;  } |

**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 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Ensures code simplicity and readability |
| PVS-Studio | 7.15 | V127 | Detects overly complex code structures |

Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Use Effective Quality Assurance Techniques | STD-010-CPP | Implement rigorous quality assurance (QA) practices such as code reviews, testing, and continuous integration to detect and mitigate defects, ensuring software reliability and security. |

| **Noncompliant Code** |
| --- |
| Process data without unit tests or code reviews |
| void processData(int\* data, int size) {  for (int i = 0; i < size; ++i) {  data[i] \*= 2;  }  } |

| **Compliant Code** |
| --- |
| Data processing with unit tests and code reviews |
| void processData(int\* data, int size) {  if (data == nullptr || size <= 0) {  return; // Handle invalid input  }    for (int i = 0; i < size; ++i) {  data[i] \*= 2;  }  } |

**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 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Cpp | Implements quality assurance techniques |
| PVS-Studio | 7.15 | V520 | Ensures rigorous testing and quality assurance practices |

**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.

**Pre-production**

1. **Assess and Plan**:
   * **Automation Integration**: Use automated tools to assess code against the defined standards. Implement static code analysis tools (like SonarQube or CodeQL) to scan the codebase for compliance with coding standards.
   * **Policy Enforcement**: Define security policies and coding standards as part of the initial project setup. This includes setting up rules for data type usage, input validation, memory management, and more.
2. **Design**:
   * **Automation Integration**: Incorporate security and compliance checks into the design phase. Use threat modeling tools to identify potential security issues early.
   * **Policy Enforcement**: Ensure that design documents and architecture include references to coding standards and security practices.
3. **Build**:
   * **Automation Integration**: Integrate automated linters and code formatters in the build pipeline to enforce coding standards.
   * **Policy Enforcement**: Use tools like Prettier and ESLint to enforce code formatting and standard adherence during the build process.
4. **Verify and Test**:
   * **Automation Integration**: Implement automated testing tools (like Jenkins or GitHub Actions) to run unit tests, integration tests, and security scans.
   * **Policy Enforcement**: Ensure that the tests include checks for compliance with coding standards. Use tools like OWASP ZAP or Burp Suite for security testing.

**Production**

1. **Transition and Health Check**:
   * **Automation Integration**: Deploy automated health checks and continuous integration/continuous deployment (CI/CD) pipelines.
   * **Policy Enforcement**: Ensure that the deployment process includes security checks and that any changes are validated against the coding standards before going live.
2. **Monitor and Detect**:
   * **Automation Integration**: Use monitoring tools like Splunk, ELK Stack, or Prometheus to continuously monitor application performance and security.
   * **Policy Enforcement**: Set up alerts for any deviations from the coding standards, such as unexpected data types or memory usage patterns.
3. **Respond**:
   * **Automation Integration**: Automate incident response with tools like PagerDuty or AWS Lambda to handle security incidents and breaches.
   * **Policy Enforcement**: Ensure that incident response plans include steps to address violations of coding standards.
4. **Maintain and Stabilize**:
   * **Automation Integration**: Regularly update and patch systems using automation tools to ensure they remain compliant with the latest standards.
   * **Policy Enforcement**: Conduct periodic reviews and audits of the codebase to ensure ongoing compliance with coding standards.

**Detailed Explanation Using the DevSecOps Diagram**

* **Assess and Plan**:
  + This stage involves evaluating the threat landscape and planning security measures. Automation can be used here to continuously scan for vulnerabilities and ensure that all new features and updates are planned with security and compliance in mind.
* **Design**:
  + During the design phase, security best practices and compliance requirements should be integrated. Automation tools can help enforce these standards by generating security requirements and validating design documents.
* **Build**:
  + The build phase should integrate automated tools for static code analysis and security scanning to enforce coding standards. Tools like SonarQube can automatically review code for adherence to standards such as proper data type usage and input validation.
* **Verify and Test**:
  + Automated testing frameworks can run unit tests, integration tests, and security scans to verify that the code complies with the defined standards. This helps in catching issues early in the development process.
* **Transition and Health Check**:
  + Automated deployment pipelines should include health checks and validation steps to ensure that the code moving to production complies with all security and coding standards.
* **Monitor and Detect**:
  + Continuous monitoring tools should be set up to detect any deviations from the standards in the production environment. These tools can generate alerts and logs for any non-compliant activities.
* **Respond**:
  + Automated incident response tools can be configured to handle security incidents swiftly, ensuring that any breaches or non-compliance issues are addressed promptly.
* **Maintain and Stabilize**:
  + Regular maintenance and updates should be automated to ensure that the application remains compliant with the latest coding standards and security requirements.

By integrating automation at each stage of the DevSecOps lifecycle, Green Pace can ensure continuous compliance with coding standards, enhancing the security and reliability of their 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-CPP | Medium | Medium | Low | Medium | 3 |
| STD-003-CPP | High | High | Medium | High | 5 |
| STD-004-CPP | Critical | High | High | Critical | 5 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Low | Low | Medium | 2 |
| STD-007-CPP | High | Medium | Medium | High | 4 |
| STD-008-CPP | Critical | High | Medium | Critical | 5 |
| STD-009-CPP | Medium | Medium | Low | Medium | 3 |
| STD-010-CPP | High | Medium | Medium | High | 4 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | All sensitive data stored on disks, databases, and backup media must be encrypted using strong encryption algorithms (e.g., AES-256). Encryption at rest protects data from unauthorized access and breaches, ensuring data confidentiality even if physical media is compromised. This policy applies to all sensitive data stored on company servers and backup devices. |
| Encryption in flight | All data transmitted over networks must be encrypted using secure protocols. Encryption in flight protects data from interception and eavesdropping during transmission. This policy applies to all data transmitted over the internet or internal networks, including emails, web traffic, and API communications. |
| Encryption in use | Sensitive data processed in memory should be encrypted where possible and protected through secure coding practices. Encryption in use ensures that sensitive data is protected even while being processed. This includes protecting data in RAM and using secure coding practices to prevent data leaks. This policy applies to all applications that handle sensitive data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users must authenticate using multi-factor authentication (MFA) to access systems and applications. Authentication ensures that only authorized users can access systems and data. MFA adds an extra layer of security by requiring additional verification steps. This policy applies to all user logins and access points. |
| Authorization | Users should have the minimum level of access necessary to perform their job functions, enforced through role-based access control. Authorization ensures that users can only access data and perform actions within their scope of responsibility. RBAC helps manage access rights efficiently and securely. This policy applies to all systems and applications where user permissions are required. |
| Accounting | All user activities must be logged and monitored, including logins, changes to the database, and file accesses. Accounting involves tracking user actions to detect and respond to suspicious activities. Logging provides an audit trail for security incidents and helps in compliance reporting. This policy applies to all critical systems and databases. |

**\***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.

**Coding Standard 1: Ensure correct and appropriate usage of data types (STD-001-CPP)**

Principles: 1, 2, 4

Justification: Ensuring correct data types aligns with principle 1 (Confidentiality), as incorrect types can lead to data leaks. Principle 2 (Integrity) is supported by maintaining data consistency. Principle 4 (Least Privilege) helps limit access to necessary data types.

**Coding Standard 2: Maintain consistent and valid values for data variables (STD-002-CPP)**

Principles: 1, 2, 4

Justification: Consistent data values support principle 1 (Confidentiality) by preventing data leaks. Principle 2 (Integrity) is upheld by ensuring data accuracy. Principle 4 (Least Privilege) limits access to modify data values.

**Coding Standard 3: Use proper string handling techniques (STD-003-CPP)**

Principles: 1, 3, 5

Justification: Proper string handling protects against buffer overflows (principle 1: Confidentiality). Principle 3 (Availability) is maintained by preventing crashes. Principle 5 (Separation of Duties) ensures that string handling responsibilities are clearly defined.

**Coding Standard 4: Prevent SQL injection vulnerabilities (STD-004-CPP)**

Principles: 1, 2, 6

Justification: Preventing SQL injections aligns with principle 1 (Confidentiality) and principle 2 (Integrity) by protecting data. Principle 6 (Defense in Depth) involves multiple layers of protection, such as input validation and parameterized queries.

**Coding Standard 5: Manage memory effectively (STD-005-CPP)**

Principles: 1, 3, 6

Justification: Effective memory management supports principle 1 (Confidentiality) by preventing memory leaks. Principle 3 (Availability) ensures system stability. Principle 6 (Defense in Depth) involves using multiple techniques to manage memory securely.

**Coding Standard 6: Use assertions to validate assumptions (STD-006-CPP)**

Principles: 2, 5, 6

Justification: Assertions support principle 2 (Integrity) by validating data. Principle 5 (Separation of Duties) ensures clear responsibility for data validation. Principle 6 (Defense in Depth) involves multiple checks to verify data integrity.

**Coding Standard 7: Handle exceptional conditions gracefully (STD-007-CPP)**

Principles: 1, 3, 6

Justification: Graceful exception handling supports principle 1 (Confidentiality) by preventing data leaks during errors. Principle 3 (Availability) ensures the system remains functional. Principle 6 (Defense in Depth) involves multiple strategies to handle errors.

**Coding Standard 8: Validate and sanitize input data (STD-008-CPP)**

Principles: 1, 2, 6

Justification: Input validation aligns with principle 1 (Confidentiality) and principle 2 (Integrity) by protecting data. Principle 6 (Defense in Depth) involves multiple layers of input checks.

**Coding Standard 9: Simplify code implementation (STD-009-CPP)**

Principles: 3, 4, 5

Justification: Simplified code supports principle 3 (Availability) by reducing errors. Principle 4 (Least Privilege) limits complex access. Principle 5 (Separation of Duties) ensures clear responsibility for code segments.

**Coding Standard 10: Implement rigorous quality assurance techniques (STD-010-CPP)**

Principles: 2, 3, 6

Justification: Quality assurance supports principle 2 (Integrity) and principle 3 (Availability) by ensuring reliable software. Principle 6 (Defense in Depth) involves thorough testing and validation.

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 05/26/2024 | Module 3 Alterations | Michael Berry | Michael Berry |
| 3.0 | 06/29/2024 | Policy Completion | Michael Berry | Michael Berry |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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