**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 | This principle emphasizes the importance of verifying and sanitizing all input data before processing it. Unvalidated input can lead to various security vulnerabilities, such as SQL injection, cross-site scripting, and buffer overflows. By ensuring that input data is valid, properly formatted, and safe, systems can prevent malicious data from causing harm or triggering unauthorized actions. |
| 1. Heed Compiler Warnings | Compiler warnings are often indicators of potential security flaws or bugs in code. By paying attention to and addressing these warnings, developers can preemptively resolve issues that might be exploited by attackers. Ignoring compiler warnings can lead to vulnerabilities in the software, as these warnings frequently point out code that doesn't adhere to best practices or might produce unexpected behavior. |
| 1. Architect and Design for Security Policies | This principle involves integrating security considerations into the architecture and design phase of software development. It means planning for secure operations, considering potential threats, and implementing security controls from the outset. A well-designed system aligns with security policies and standards, ensuring that security is not an afterthought but a foundational component. |
| 1. Keep It Simple | Complexity often leads to security vulnerabilities. The principle of keeping it simple advocates for straightforward, easy-to-understand, and maintainable designs and code. Simple systems are easier to analyze for security flaws, and their behavior under various conditions is more predictable, reducing the likelihood of security bugs. |
| 1. Default Deny | This security principle states that a system should deny all access by default and only grant permissions and resources to users, programs, or processes that are explicitly allowed. This approach minimizes the attack surface and ensures that only authorized entities have access, reducing the risk of unauthorized or malicious access. |
| 1. Adhere to the Principle of Least Privilege | This principle dictates that users, programs, and systems should have the minimum level of access necessary to perform their tasks. By limiting permissions and access rights, the potential damage from a security breach can be minimized, as attackers or compromised accounts have restricted capabilities within the system. |
| 1. Sanitize Data Sent to Other Systems | When data is transferred to external systems, it should be sanitized to prevent the propagation of security threats. This includes removing sensitive information, stripping potentially malicious content, and ensuring that the data conforms to the expected format of the receiving system. Sanitization protects against attacks like SQL injection and data leakage. |
| 1. Practice Defense in Depth | This principle involves using multiple layers of security controls and practices to protect systems and data. By implementing diverse defensive strategies, an attacker must bypass several barriers, reducing the likelihood of a successful breach. Defense in depth includes physical security, network security, application security, and user education, among others. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance in software development includes rigorous testing and review processes to identify and fix security vulnerabilities and bugs. Effective QA techniques involve a combination of automated testing tools, code reviews, penetration testing, and other methodologies to ensure that the software is robust and secure against known and potential threats. |
| 1. Adopt a Secure Coding Standard | This principle emphasizes the importance of following established secure coding guidelines and standards. These standards provide a framework for writing code that is resilient to common security vulnerabilities. Adhering to these standards helps developers avoid known pitfalls and ensures a consistent approach to security across the software development lifecycle. |

### 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] | **Data Type Integrity** - Ensures that data types are used properly to prevent type-related errors and improve code maintainability. |

| **Noncompliant Code** |
| --- |
| Using an integer type where a floating point is expected can lead to precision errors. |
| int calculateArea(int length, int width) {  return length \* width;  } |

| **Compliant Code** |
| --- |
| Use floating-point types for calculations requiring decimal precision. |
| double calculateArea(double length, double width) {  return length \* width;  } |

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

| **Principle**: Adopt a Secure Coding Standard **Explanation**: Ensuring data types are used properly aligns with following secure coding standards, as it helps avoid type-related errors and improves code maintainability, directly preventing common vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | Type Mismatch | Detects misuse of data types leading to potential type-related errors. |
| Coverity | Latest | BAD\_COMPARE | Identifies incorrect data type comparisons. |
| PVS-Studio | Latest | Viva64 | Detects potential type conversion and compatibility issues in C++ code. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | **Data Value Integrity** -Ensuring that data values are initialized and used correctly to prevent undefined behaviors. |

| **Noncompliant Code** |
| --- |
| Using an uninitialized variable. |
| int value;  std::cout << value; // Undefined behavior |

| **Compliant Code** |
| --- |
| Initializing variables before use. |
| int value = 0;  std::cout << value; // Defined behavior |

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

| **Principle**: Use Effective Quality Assurance Techniques **Explanation**: Initializing variables before use is a practice that can be ensured through rigorous QA techniques, including static analysis and code reviews, to prevent undefined behaviors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | Latest | Memcheck | Identifies uninitialized memory reads. |
| Coverity | Latest | UNINIT | Detects use of uninitialized variables. |
| CPPCheck | Latest | uninitvar | Static analysis tool to find variables that are used uninitialized. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | **String Handling** - Proper handling of strings to prevent buffer overflows and other string-related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Unsafe string copy. |
| char dest[10];  strcpy(dest, "This is a very long string"); |

| **Compliant Code** |
| --- |
| Using safe string functions. |
| char dest[10];  strncpy(dest, "Short", sizeof(dest));  dest[sizeof(dest) - 1] = '\0'; |

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

| **Principle**: Validate Input Data **Explanation**: Proper handling of strings, especially those received as input, is crucial for preventing buffer overflows and other string-related vulnerabilities, embodying the principle of validating all input data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | Latest | STRING\_OVERFLOW | Detects potential buffer overflows in string operations. |
| Fortify | Latest | Buffer Overflow | Identifies unsafe string manipulation that may lead to overflows. |
| Clang Static Analyzer | Latest | strcpy\_chk | Checks for dangerous string copy operations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **Preventing SQL Injection** - Secure coding practices to prevent SQL injection vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Concatenating user input in SQL query. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userName + "'"; |

| **Compliant Code** |
| --- |
| Using parameterized queries. |
| std::string query = "SELECT \* FROM users WHERE name = ?";  // Use a method to bind 'userName' to the query |

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

| **Principle**: Sanitize Data Sent to Other Systems **Explanation**: Using parameterized queries to prevent SQL injection is an application of data sanitization, ensuring that data sent to the database is safe and free from malicious content. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | Latest | SQL Injection | Identifies SQL injection vulnerabilities. |
| Coverity | Latest | SQL\_INJECTION | Detects concatenations that could lead to SQL injection. |
| SonarQube | Latest | sql-injection | Static code analysis to prevent SQL injection vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **Memory Management** - Ensuring safe and effective memory management to prevent leaks and corruption. |

| **Noncompliant Code** |
| --- |
| Improper memory deallocation. |
| int\* ptr = new int(10);  // Missing delete for ptr |

| **Compliant Code** |
| --- |
| Correct memory deallocation. |
| int\* ptr = new int(10);  delete ptr; |

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

| **Principle**: Keep It Simple **Explanation**: Ensuring safe and effective memory management reduces complexity and the likelihood of errors such as memory leaks and corruption, aligning with the principle of simplicity in design and implementation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | Latest | Memcheck | Identifies memory leaks and incorrect memory deallocation. |
| Coverity | Latest | RESOURCE\_LEAK | Detects potential memory leaks and improper memory management. |
| Clang Static Analyzer | Latest | alpha.unix.Malloc | Checks for memory allocation/deallocation mismatches. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Use of Assertions** - Proper use of assertions for debugging and validating assumptions. |

| **Noncompliant Code** |
| --- |
| Misusing assertions for regular error handling. |
| assert(ptr != nullptr && "Pointer must not be null"); |

| **Compliant Code** |
| --- |
| Using assertions for debug-only checks. |
| #ifndef NDEBUG  assert(ptr != nullptr && "Pointer must not be null");  #endif |

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

| **Principle**: Heed Compiler Warnings **Explanation**: Proper use of assertions, especially in debug modes, can help identify issues early in development, and paying attention to compiler warnings related to assertions can prevent misuse. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | Latest | assertUsed | Verifies assertions are used correctly and not for error handling. |
| Clang Static Analyzer | Latest | debug.AssertMisuse | Checks for misuse of assertions in non-debug code. |
| Coverity | Latest | ASSERT\_SIDE\_EFFECT | Identifies assertions that cause side effects. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | **Exception Handling** - Ensuring exceptions are used and handled correctly to maintain program stability. |

| **Noncompliant Code** |
| --- |
| Throwing exceptions from a destructor. |
| class MyClass {  public:  ~MyClass() {  throw std::runtime\_error("Error in destructor");  }  }; |

| **Compliant Code** |
| --- |
| Avoid throwing exceptions from destructors. |
| class MyClass {  public:  ~MyClass() noexcept {  // Clean up resources without throwing exceptions  }  }; |

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

| **Principle**: Architect and Design for Security Policies **Explanation**: Incorporating proper exception handling into the system's design ensures that it is robust against unexpected conditions, aligning with the principle of considering security from the outset. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | Latest | UNCAUGHT\_EXCEPT | Identifies exceptions that are thrown but not caught. |
| CPPCheck | Latest | nothrow | Checks for functions that should not throw exceptions but do. |
| Clang Static Analyzer | Latest | UnhandledException | Analyzes code paths for potential unhandled exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Validation** | STD-008-CPP | **Input Validation** - Ensuring all user input is validated to prevent injection attacks, buffer overflows, and other input-based vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Accepting and using user input without validation. |
| char buffer[50];  std::cin >> buffer; // Potential buffer overflow if input exceeds 50 characters |

| **Compliant Code** |
| --- |
| Validating user input to ensure it meets expected criteria. |
| char buffer[50];  std::cin.getline(buffer, sizeof(buffer));  // Additional checks for buffer content |

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

| **Principle**: Validate Input Data **Explanation**: Validating all user input before processing is fundamental to securing applications against various forms of input-based vulnerabilities, directly applying the principle of input validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | Latest | Unvalidated Input | Identifies inputs that are not properly validated before use. |
| Coverity | Latest | TAINTED\_DATA | Analyzes for potential vulnerabilities from unvalidated user input. |
| SonarQube | Latest | input-validation | Static analysis to enforce input validation. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Error Handling and Logging** | STD-009-CPP | **Error Handling and Logging** - Ensuring that errors are handled gracefully and logged appropriately to maintain the stability and security of the software. |

| **Noncompliant Code** |
| --- |
| Ignoring potential errors returned by functions. |
| FILE\* file = fopen("example.txt", "r");  // Ignoring the case where file might be NULL |

| **Compliant Code** |
| --- |
| Checking and handling error conditions effectively. |
| FILE\* file = fopen("example.txt", "r");  if (file == nullptr) {  std::cerr << "Error opening file" << std::endl;  // Handle error appropriately  } |

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

| **Principle**: Use Effective Quality Assurance Techniques **Explanation**: Ensuring that errors are handled gracefully and logged appropriately can be achieved through comprehensive QA practices, including testing for error conditions and reviewing log management strategies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | error-handling | Ensures errors are handled and logged properly. |
| Coverity | Latest | CHECKED\_RETURN | Verifies that all error codes are checked and handled correctly. |
| Fortify | Latest | Poor Error Handling | Identifies insufficient error handling patterns. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Use of Smart Pointers** | STD-010-CPP | **Smart Pointer Usage** - Promoting the use of smart pointers to automatically manage memory, thus reducing the risk of memory leaks and pointer-related errors. |

| **Noncompliant Code** |
| --- |
| Manual memory management with raw pointers. |
| int\* ptr = new int(10);  // Some code...  delete ptr; // Risk of forgetting to delete or double delete |

| **Compliant Code** |
| --- |
| Utilizing smart pointers for automatic memory management. |
| std::unique\_ptr<int> ptr(new int(10));  // Memory is automatically managed |

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

| **Principle**: Adopt a Secure Coding Standard **Explanation**: The use of smart pointers for memory management is recommended by many secure coding standards as it automatically reduces the risk of memory-related issues, demonstrating adherence to established best practices for secure coding. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | SmartPtr | Checks for correct use of smart pointers to manage memory. |
| Coverity | Latest | AUTOSAR.MEM.MEM007 | Enforces rules for smart pointer usage per AUTOSAR standards. |
| CPPCheck | Latest | autoMemoryLeak | Identifies potential memory leaks that could be prevented with smart pointers. |

### 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.

In the DevSecOps lifecycle, automation plays a crucial role in ensuring that coding standards are enforced, and compliance is maintained from the early stages of development to production. Within the pre-production phase, automation should be integrated at multiple points. Initially, during the "Assess and Plan" and "Design" stages, automated tools can be used to review design documents and code for compliance with secure coding standards, ensuring that security considerations are incorporated from the outset. As we move to the "Build" stage, automation tools such as static code analyzers and linters can be integrated into the Continuous Integration (CI) pipeline to automatically scan code commits for violations of the defined coding standards. This ensures immediate feedback to developers and prevents non-compliant code from progressing further down the development pipeline.

Further into the "Verify and Test" stage, dynamic analysis tools and automated testing frameworks can be employed to test the application against known vulnerabilities and compliance requirements, offering a more in-depth assessment of the application's security posture before it transitions to production. In the production phase, automation shifts towards monitoring and maintaining compliance. During the "Monitor and Detect" stage, automated monitoring tools can continuously scan for security issues and compliance drift, while in the "Respond" stage, automated response mechanisms can be implemented to address detected issues in real-time. Finally, throughout the "Maintain and Stabilize" stage, automation supports the ongoing assessment of security posture and compliance with coding standards, ensuring that any updates or changes to the application remain within compliance boundaries. Integrating automation throughout this lifecycle not only streamifies the compliance process but also embeds security into the DNA of the development and operational processes, enhancing the overall security and integrity of Green Pace's software offerings.

### 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 | Medium | Medium | High | 2 |
| STD-002-CPP | High | High | Low | High | 4 |
| STD-003-CPP | High | Medium | Medium | High | 3 |
| STD-004-CPP | High | Medium | Medium | High | 3 |
| STD-005-CPP | High | Medium | High | High | 4 |
| STD-006-CPP | Medium | Low | Low | Medium | 2 |
| STD-007-CPP | High | Medium | Medium | High | 3 |
| STD-008-CPP | High | High | Medium | High | 4 |
| STD-009-CPP | Medium | Medium | Medium | Medium | 2 |
| STD-010-CPP | High | Medium | Medium | 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 in rest | Encryption at rest involves transforming data into a format that is unreadable without the decryption key. This type of encryption is critical for protecting stored data, such as databases, files, and backups, from unauthorized access. The policy applies to all stored data, regardless of its storage location (on-premises or cloud environments). It should be applied by encrypting hard drives, databases, and backups with strong encryption algorithms and securely managing the encryption keys. This policy is used to ensure that data is protected against breaches and theft when it is not being actively accessed or used. |
| Encryption at flight | Encryption in flight is the process of encrypting data while it is being transmitted over a network. This policy applies to all data transmitted between systems, clients, and servers to protect it from interception and eavesdropping. It should be implemented using protocols such as TLS (Transport Layer Security) for web traffic, SSH (Secure Shell) for remote administration, and VPNs (Virtual Private Networks) for secure remote access. Encrypting data in flight is essential for maintaining the confidentiality and integrity of data as it moves across networks, particularly over unsecured or public networks. |
| Encryption in use | Encryption in use involves encrypting data that is being actively processed by applications. This type of encryption protects data that is in memory or during computation processes, ensuring it remains secure even in a compromised runtime environment. The policy applies to sensitive operations and data, such as processing payment transactions or personal information, where data must be protected from unauthorized access, including access from system administrators and cloud providers. Implementing encryption in use requires adopting technologies like homomorphic encryption or secure enclaves (e.g., Intel SGX) to enable computations on encrypted data without exposing it in plaintext. This policy is crucial for protecting sensitive data against memory dumps, malware, and insider threats. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users before they gain access to systems or data. This policy is mandatory for all user logins, employing strong authentication mechanisms such as passwords, multi-factor authentication (MFA), or biometrics. By verifying user identities rigorously, the organization ensures that only authorized individuals can access its systems, protecting against unauthorized access attempts. This process is critical at every entry point, from databases to applications and network systems, ensuring the integrity and confidentiality of the data and systems. |
| Authorization | Authorization determines and enforces the actions and resources a user can access after being authenticated. This policy directly addresses user level of access and the files accessed by users, ensuring that individuals can only engage with data and perform actions that are appropriate to their roles within the organization. It is applied using role-based access control (RBAC) or attribute-based access control (ABAC) systems, ensuring a user's access rights are in line with their job requirements. Changes to the database, including the addition of new users, also fall under this policy, requiring strict controls to ensure that access permissions are correctly assigned and managed. |
| Accounting | Accounting, or auditing, involves tracking and logging user activities within systems and databases, including user logins, changes to the database, additions of new users, user levels of access, and files accessed by users. This policy ensures that all significant actions are recorded, creating an auditable trail of activities for security monitoring, compliance, and forensic analysis. Implementing a comprehensive logging mechanism and regular review of logs are essential for detecting unauthorized or suspicious activities, ensuring accountability, and maintaining the integrity of the system. Automated tools should be employed to facilitate the monitoring and analysis of logged data, enabling timely responses to potential security incidents. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 02/05/2024 | Standards Update/Automation Considerations | Hunter Richards |  |
| 1.2 | 02/07/2024 | Framework Policies/Policy Justifications | Hunter Richards |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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