**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 | All input data must be validated to prevent injection attacks, buffer overflows, and other exploits. Unvalidated input can lead to serious vulnerabilities such as SQL injections or cross site scripting (CSS). |
| 1. Heed Compiler Warnings | Compiler warnings can indicate potential security flaws. Developers must address all warnings by either resolving the issues or explicitly documenting justifications for ignoring them. |
| 1. Architect and Design for Security Policies | Security must be integrated into the architecture and design phase to ensure that applications are resilient against attacks. This includes secure authentication, encryption, and access controls. |
| 1. Keep It Simple | Complex security mechanisms can introduce vulnerabilities. Security implementations should be as simple as possible while maintaining robustness. |
| 1. Default Deny | Systems should deny access by default and only grant permissions when explicitly authorized. This reduces the risk of unintended access. |
| 1. Adhere to the Principle of Least Privilege | Users and processes should only have the minimum level of access necessary to perform their tasks. This reduces the risk of exploitation |
| 1. Sanitize Data Sent to Other Systems | Data should be sanitized before being sent to external systems to prevent security breaches. This includes encoding, escaping, and validating data. |
| 1. Practice Defense in Depth | Multiple layers of security should be implemented to mitigate risks. This includes using firewalls, intrusion detection, and validating data. |
| 1. Use Effective Quality Assurance Techniques | Code should undergo rigorous testing, including static analysis, penetration testing, and code reviews, to identify vulnerabilities. |
| 1. Adopt a Secure Coding Standard | A defined secure coding standard should be followed to ensure consistency and security in the codebase |

### 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 non-compliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-DTS] | Improper use of data types can lead to buffer overflows and memory corruption. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses *strcpy*, which does not check the size of the destination buffer, potentially leading to buffer overflows. |
| char buffer[10];  strcpy(buffer, userInput); // Unsafe |

| **Compliant Code** |
| --- |
| The compliant code uses *strncpy* with a size check and null termination to prevent buffer overflows. |
| char buffer[10];  strn00cpy(buffer, userInput, sizeof(buffer) – 1);  buffer[sizeof(buffer) -1] = ‘\0’; // Safe |

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

| **Principles(s):** **1. Validate Input Data**: Ensures data is validated before use. **4. Keep It Simple**: Use simpler, safer functions like *strncpy* over *strcpy*. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1213 | Detects improper use of unsafe string functions like *strcpy*. |
| Checkmarx | 9.3 | CX-2060 | Identifies unsafe buffer handling or improper data type usage. |
| Coverity | 2021.4 | CWE-120 | Identifies buffer overflows and unsafe memory operations. |
| Veracode | 2021.3 | VU-3050 | Detects memory corruption risks and improper data type use. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DVS] | Hardcoded credentials or unchecked values can lead to security vulnerabilities |

| **Noncompliant Code** |
| --- |
| The noncompliant code hardcodes a password, making it easily discoverable and exploitable |
| char password[] = "admin123"; // Hardcoded password |

| **Compliant Code** |
| --- |
| The compliant code gets the password from user input, reducing the risk pf hardcoded credentials. |
| char password[MAX\_LEN];  getUserInput(password, MAX\_LEN); // Secure |

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

| **Principles(s):** **6. Adhere to the Principle of Least Privilege**: Prevent using hardcoded passwords, ensure users provide secure credentials. **5. Default Deny**: Ensures credentials are provided by the user rather than being hardcoded. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 9.3 | CX-2110 | Identifies hardcoded credentials or unsafe value handling. |
| SonarQube | 8.9 | S1234 | Detects hardcoded values like passwords or tokens in codebase. |
| Fortify | 20.2 | SEC-3050 | Detects sensitive data hardcoded in the code or configuration files. |
| Veracode | 2021.3 | VU-4000 | Flags unprotected hardcoded credentials that need to be secured. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-SCS] | Unchecked string operations can cause buffer overflows. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses *scanf* without a size limit, which can lead to buffer overflows if the input exceeds the buffer size. |
| char name[20];  scanf("%s", name); // Unsafe |

| **Compliant Code** |
| --- |
| The compliant code uses *scanf* with a size limit (*%19s*), preventling buffer overflows by restricting input length. |
| char name[20];  scanf("%19s", name); // Safe |

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

| **Principles(s):** **1. Validate Input Data**: Ensure inputs conform to expected size constraints. **2. Heed Compiler Warnings**: Compiler warnings for unsafe string functions (e.g., scanf) should be addressed to prevent vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.4 | CWE-120 | Detects unchecked string operations and potential overflows. |
| SonarQube | 8.9 | S1141 | Flags unsafe string functions that don’t account for buffer size. |
| Checkmarx | 9.3 | CX-3070 | Identifies buffer overflow risks with unprotected string functions. |
| Fortify | 20.2 | SEC-4090 | Scans for unsafe string handling like *scanf* and improper bounds checking. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | Directly concatenating user input in SQL queries leads to SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| The noncompliant code directly concatenates user input into an SQL query, making it vulnerable to SQL injection attacks. |
| sprintf(query, "SELECT \* FROM users WHERE name = '%s'", userInput); |

| **Compliant Code** |
| --- |
| The compliant code uses prepared statements and parameter binding, which prevent SQL injection by treating user input as data, not executable code. |
| stmt = prepareStatement("SELECT \* FROM users WHERE name = ?");  bindParameter(stmt, userInput); |

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

| **Principles(s):** **1. Validate Input Data**: Sanitize user input to prevent SQL injection. **3. Architect and Design for Security**: Design the application with secure methods like parameterized queries to prevent SQL injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1788 | Flags SQL injection risks due to direct concatenation of user input. |
| Checkmarx | 9.3 | CX-3000 | Detects SQL injection by analyzing. |
| Coverity | 2021.4 | CWE-89 | Identifies SQL injection vulnerabilities due to improper parameter binding. |
| Fortify | 20.2 | SEC-5010 | Detects potential SQL injection vulnerabilities in SQL string concatenation. |

#### 

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MPS] | Improper memory management can lead to leaks and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The noncompliant code lacks proper checks after *malloc* and doesn’t handle potential allocation failures and does not limit the copy of user input. |
| char \*ptr = malloc(100);  strcpy(ptr, userInput);  free(ptr);  ptr = NULL; |

| **Compliant Code** |
| --- |
| The compliant code checks the results of *malloc,* uses *strncpy* with a size limit, and null terminates the string, ensuring memory safety. |
| char \*ptr = malloc(100);  if (ptr) {  strncpy(ptr, userInput, 99);  ptr[99] = '\0';  free(ptr);  ptr = NULL;  } |

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

| **Principles(s):** **2. Heed Compiler Warnings**: Warnings about potential memory allocation issues should be addressed. **4. Keep It Simple**: Use memory-safe functions like *strncpy* and handle errors for malloc. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.4 | CWE-120 | Detects unsafe memory access and improper memory handling. |
| SonarQube | 8.9 | S1440 | Detects improper handling of dynamically allocated memory. |
| Checkmarx | 9.3 | CX-4000 | Flags unprotected memory access or lack of memory allocation checks. |
| Fority | 20.2 | SEC-5070 | Identifies risks related to improper memory management (e.g., unprotected malloc calls). |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASS] | Assertions help detect logical errors but should not be used for security checks. |

| **Noncompliant Code** |
| --- |
| The noncompliant code misuses *assert* for security checks, which can be disabled in production, leaving vulnerabilities. |
| assert(userInput != NULL); // Misuse of assertion |

| **Compliant Code** |
| --- |
| The compliant code uses a regular *if* statement for error checking, ensuring the check is always performed. |
| if (userInput == NULL) {  return ERROR\_CODE;  } |

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

| **Principles(s):** **4. Keep It Simple**: Regular error-checking mechanisms (e.g., if statements) should be used in place of assertions. **9. Use Effective Quality Assurance Techniques**: Ensure proper validation and error handling by using non-assertive checks in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1102 | Flags the misuse of assertions for security checks. |
| Checkmarx | 9.3 | CX-1050 | Detects assertions used for critical security checks. |
| Coverity | 2021.4 | CWE-675` | Identifies assertions used inappropriately for critical checks that could be disabled. |
| Vercode | 2021.3 | VU-7025 | Flags the use of assertions in security-sensitive areas |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EHS] | Proper exception handling ensures graceful failure. |

| **Noncompliant Code** |
| --- |
| The noncompliant code doesn’t handle potential errors from *riskyFunction*, leading to unhandled exceptions or unexpected behavior. |
| int result = riskyFunction();  printf("Success"); |

| **Compliant Code** |
| --- |
| The compliant code checks the return value of *riskyFunction* and handles errors appropriately. |
| int result = riskyFunction();  if (result != SUCCESS) {  handleError();  } |

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

| **Principles(s):** **9. Use Effective Quality Assurance Techniques**: Proper exception handling ensures that the application can gracefully recover from errors. **4. Keep It Simple**: Use simple error checks instead of complex control structures like assertions for handling errors |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1300 | Flags missing exception handling for risky operations. |
| Checkmarx | 9.3 | CX-2100 | Detects unhandled exceptions in risky functions. |
| Coverity | 2021.4 | CWE-398 | Identifies missing error handling for system calls. |
| Fortify | 20.2 | SEC-3030 | Flags failure to properly handle exceptions that could lead to unhandled errors. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Sensitive Data Erasure** | [STD-008-SMH] | Sensitive data should be securely erased from memory after use. |

| **Noncompliant Code** |
| --- |
| The noncompliant code does not clear the password from memory after use, potentially leaving sensitive data exposed. |
| char password[MAX\_LEN];  getUserInput(password, MAX\_LEN);  // Password remains in memory after this point |

| **Compliant Code** |
| --- |
| The compliant code use *memset* to securely erase sensitive data from memory after use, preventing potential exposure. |
| memset(password, 0, sizeof(password)); |

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

| **Principles(s):** **7. Sanitize Data Sent to Other Systems**: Ensure that sensitive data is erased securely after use, preventing leaks. **8. Practice Defense in Depth**: Multiple layers of protection, including secure memory erasure, ensure that data is not exposed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1123 | Flags instances where sensitive data is not properly erased from memory |
| Checkmarx | 9.3 | CX-3005 | Detects failure to securely erase sensitive data from memory after use. |
| Coverity | 2021.4 | CWE-214 | Flags memory leakage or failure to securely clear sensitive data |
| Fortify | 20.2 | SEC-4030 | Detects failure to erase passwords or sensitive info after use. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Random Number Generation** | [STD-009-RNG] | Using predictable random numbers can compromise security. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses *rand(),* which is pseudo-random number generator and is not cryptographically secure, leading to predictable outputs. |
| char buffer[10];  for(int i = 0; i < sizeof(buffer); i++) {  buffer[i] = rand() % 256;  } |

| **Compliant Code** |
| --- |
| The compliant code uses *RAND\_bytes,* a cryptographically secure random number generator, to prevent predictable random numbers. |
| RAND\_bytes(buffer, sizeof(buffer)); |

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

| **Principles(s):** **6. Adhere to the Principle of Least Privilege**: Using cryptographically secure random numbers ensures that secrets and keys cannot be easily predicted. **4. Keep It Simple**: Use secure and simple random number generation methods instead of rand(), which is not cryptographically secure. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1124 | Detects use of non-cryptographically secure random number generators like *rand()* |
| Checkmarx | 9.3 | CX-3040 | Flags usage of weak random number generators in cryptographic contexts. |
| Coverity | 2021.4 | CWE-330 | Identifies weak random number generation in security sensitive code. |
| Fortify | J20.2 | SEC-7010 | Detects use of pseudo-random generators in security applications. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Race Conditions** | [STD-010-RCE] | Race conditions can lead to unpredictable behavior and security issues. |

| **Noncompliant Code** |
| --- |
| The noncompliant code accesses and modifies a shared resource without any synchronization mechanism, leading to potential race conditions. |
| sharedResource = newValue; // Unprotected access |

| **Compliant Code** |
| --- |
| The compliant code uses a mutex lock to protect shared resources, preventing race conditions. |
| pthread\_mutex\_lock(&lock);  sharedResource = newValue;  pthread\_mutex\_unlock(&lock); |

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

| **Principles(s): 4. Keep It Simple**: Use mutexes for simple protection of shared resources to prevent complex race conditions. **8. Practice Defense in Depth**: Locking mechanisms add an additional layer of protection to avoid race conditions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | S1110 | Detects unsynchronized access to shared resources that could lead to race conditions. |
| Checksmarx | 9.3 | CX-5000 | Flags access to shared resources without synchronization mechanisms. |
| Coverity | 2021.4 | CWE-362 | Detects possible race conditions due to improper resource locking |
| Fortity | 20.2 | SEC-9030 | Identifies lack of mutex or lock mechanisms in concurrent resource access. |

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.

By embedding security measures into every stage of the DevOps toolchain, DevOps evolves into DevSecOps, ensuring security is a foundational aspect throughout the software development lifecycle. During the **“Assess and Plan”** phase, automated threat modeling and risk assessments are utilized to proactively identify vulnerabilities and potential security risks. In the **“Design”** and **“Build”** phases, security is integrated through IDE security checks, enforcing best practices early in the development process.

In the **“Verify & Test”** phase, static code analysis, dynamic application security testing, and automated penetration tests are conducted in conjunction with standard unit and integration tests to uncover and address any security weaknesses.

Once in **Production**, security measures remain integral through continuous monitoring, integrity checks, and layered defense strategies. Techniques such as network monitoring, performance logging, and ongoing penetration testing are employed to detect and mitigate new threats. Just as quality assurance testing is performed consistently, security testing should also be continuous, ensuring that vulnerabilities are identified and remediated at the earliest possible stage.

### 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-DVS | High | Likely | High | Medium | 3 |
| STD-003-SCS | High | Likely | Medium | High | 3 |
| STD-004-SQL | High | Unlikely | Medium | High | 2 |
| STD-005-MPS | High | Likely | Medium | High | 2 |
| STD-006-ASS | Medium | Likely | Low | Medium | 4 |
| STD-007-EHS | High | Unlikely | High | High | 1 |
| STD-008-SMH | High | Unlikely | Medium | High | 2 |
| STD-009-RNG | High | Unlikely | High | High | 1 |
| STD-010-RCE | High | Likely | High | High | 1 |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-DVS | High | Likely | High | Medium | 3 |
| STD-003-SCS | High | Likely | 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 at rest | Encryption at rest refers to the protection of data stored on physical media when it is not in active use. This ensures that sensitive data remains protected from unauthorized access in case of theft or a data breach. The policy applies when storing sensitive data, such as financial records or personal information, ensuring that it is encrypted using secure algorithms like AES. It should be applied in all storage solutions where sensitive data resides. |
| Encryption in flight | Encryption in flight secures data while it is being transmitted across networks. This prevents unauthorized access during transmission. The policy applies to all communications of sensitive information, requiring encryption protocols like TLS/SSL to protect data between users and systems. It is mandatory for all transmissions involving sensitive data, ensuring confidentiality and data integrity over public and private networks. |
| Encryption in use | Encryption in use ensures that data remains protected while it is being actively processed or accessed in memory. Unlike encryption at rest or in flight, this protects data during active use by applications and services. The policy applies to high-risk systems where sensitive data is processed, such as in-memory encryption for database queries or real-time analytics. It is essential to apply this policy when handling live data in applications. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of users or systems attempting to access secured resources. The policy applies whenever a user or system tries to access sensitive systems, requiring methods such as multi-factor authentication (MFA) or digital certificates. Strong authentication policies ensure that only authorized users are granted access, reducing the risk of unauthorized access and data breaches. |
| Authorization | Authorization controls what an authenticated user or system is allowed to do within the system. The policy applies after a user is authenticated and defines access levels based on roles or permissions. It ensures that users can only access and perform tasks relevant to their job function, following the principle of least privilege. This policy is crucial for safeguarding sensitive data and systems by restricting access according to defined roles. |
| Accounting | Accounting involves tracking and logging user actions and access within the system, enabling administrators to monitor and audit user activity. This policy applies to all systems that store or manage sensitive data, requiring detailed logs of user actions such as login attempts, file access, and changes to data. Accounting ensures that there is a traceable record of user activities, which is vital for detecting unauthorized access, performing audits, and meeting compliance standards like GDPR or HIPAA. |

**\***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 | 03/24/2025 | Module 3 Milestone | Marc Aradillas | [Insert text.] |
| [Insert text.] | 04/16/2025 | Module 6 | Marc Aradillas | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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