**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 3](#_Toc52464053)

[Purpose 3](#_Toc52464054)

[Scope 3](#_Toc52464055)

[Module Three Milestone 3](#_Toc52464056)

[Ten Core Security Principles 3](#_Toc52464057)

[C/C++ Ten Coding Standards 4](#_Toc52464058)

[Coding Standard 1 5](#_Toc52464059)

[Coding Standard 2 7](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 21](#_Toc52464067)

[Coding Standard 10 23](#_Toc52464068)

[Defense-in-Depth Illustration 25](#_Toc52464069)

[Project One 25](#_Toc52464070)

[1. Revise the C/C++ Standards 25](#_Toc52464071)

[2. Risk Assessment 25](#_Toc52464072)

[3. Automated Detection 26](#_Toc52464073)

[4. Automation 26](#_Toc52464074)

[5. Summary of Risk Assessments 27](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 27](#_Toc52464076)

[7. Map the Principles 29](#_Toc52464077)

[Audit Controls and Management 31](#_Toc52464078)

[Enforcement 31](#_Toc52464079)

[Exceptions Process 31](#_Toc52464080)

[Distribution 33](#_Toc52464081)

[Policy Change Control 33](#_Toc52464082)

[Policy Version History 33](#_Toc52464083)

[Appendix A Lookups 33](#_Toc52464084)

[Approved C/C++ Language Acronyms 33](#_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 received from users, external systems, or other sources should be considered untrusted until properly validated. Validating input ensures it adheres to expected formats, ranges, and types to prevent issues like buffer overflows, SQL injection, or other exploits that rely on malformed or malicious data. |
| 1. Heed Compiler Warnings | Compiler warnings often signal potential security risks or logic flaws in code. Developers should treat warnings seriously and resolve them immediately rather than ignore or suppress them, as they can reveal vulnerabilities like type mismatches, uninitialized variables, or unsafe memory use. |
| 1. Architect and Design for Security Policies | Security should be built into the system’s design from the beginning, not added later as an afterthought. Developers must understand the organization’s security policies and design software that supports access control, data protection, and risk mitigation from the architectural level. |
| 1. Keep It Simple | Simple, clear, and well-structured code is less likely to contain hidden bugs or security flaws. Complex logic can obscure vulnerabilities and make it harder to test or audit for security. Following the principle of simplicity helps reduce attack surfaces and improve maintainability. |
| 1. Default Deny | Systems should be designed to deny access or actions by default unless explicitly permitted. This approach minimizes the chance of unauthorized access due to misconfiguration or overlooked edge cases, enforcing stricter security by requiring deliberate allowance of operations. |
| 1. Adhere to the Principle of Least Privilege | Each part of the software should operate using the minimum privileges necessary to perform its function. Limiting access reduces the impact of a breach or misuse, ensuring that if one component is compromised, it does not gain broader control over the system. |
| 1. Sanitize Data Sent to Other Systems | Data sent from your application to other systems, such as databases, APIs, or web browsers, should be sanitized to remove or neutralize potentially harmful input. This prevents injection attacks and ensures your software does not become a vector for attacking other systems. |
| 1. Practice Defense in Depth | No single security measure is foolproof. Multiple layers of defense, such as input validation, authentication, access control, and encryption should be used in tandem so that if one mechanism fails, others still protect the system. This layered approach improves resilience. |
| 1. Use Effective Quality Assurance Techniques | Security flaws often stem from coding errors or oversight. Thorough testing methods like code reviews, static analysis, and fuzz testing help identify bugs early. Incorporating secure coding practices into QA ensures vulnerabilities are caught before deployment. |
| 1. Adopt a Secure Coding Standard | Using a recognized secure coding standard, such as the SEI CERT C++ guidelines, helps developers follow consistent, vetted best practices. These standards provide rules and recommendations that reduce vulnerabilities and improve software quality and maintainability. |

### 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-CPL | Avoid mixing different data types in expressions. Mixing different data types (e.g., signed and unsigned integers) in expressions can lead to unintended behavior due to implicit type conversions. These conversions can cause loss of precision, overflows, or incorrect comparisons, potentially resulting in vulnerabilities or logic errors. Following this standard ensures code behaves predictably and is easier to maintain and secure. |

| **Noncompliant Code** |
| --- |
| This code mixes signed and unsigned types, which can cause incorrect logic flow when negative numbers are involved. |
| int balance = -100;  unsigned int threshold = 50;  if (balance < threshold) {  std::cout << "Below threshold" << std::endl;  } |

| **Compliant Code** |
| --- |
| This code ensures both operands in the comparison are of the same type by explicitly casting to a signed type. |
| int balance = -100;  int threshold = 50;  if (balance < threshold) {  std::cout << "Below threshold" << std::endl;  } |

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

| **Principles(s):** Validate Input Data, Adopt a Secure Coding Standard  This standard aligns with the principles of validating input (to ensure type-safe operations) and adopting a secure coding standard by preventing unexpected behavior due to implicit conversions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | Latest | MISRA.CAST.TYPE | Detects unsafe type casts |
| SonarQube | 10.x | cpp:S6326 | Warns on implicit signed/unsigned conversions |
| Clang-Tidy | 17.0 | bugprone-signed-char-misuse | Flags signed/unsigned issues |
| CodeSonar | Latest | TypeConversionChecker | Checks for implicit type conversion errors |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validate Data Values Before Use. Validating data values before using them ensures that variables contain expected and safe information. This helps prevent undefined behavior, logic errors, and vulnerabilities like division by zero or using uninitialized memory. By checking data against expected constraints, we can reduce the risk of system crashes or security breaches. |

| **Noncompliant Code** |
| --- |
| Fails to check if the divisor is zero before performing division, which could lead to a runtime error or undefined behavior. |
| int divide(int a, int b) {  return a / b;  } |

| **Compliant Code** |
| --- |
| Validates that the divisor is not zero before performing the division, preventing runtime errors or crashes. |
| int divide(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Divisor cannot be zero");  }  return a / b;  } |

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

| **Principles(s):** Validate Input Data. This principle applies because it ensures that input values (like divisors) are within a valid and safe range before being used, reducing the risk of logic errors and vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 15.0 | bugprone-divide-by-zero | Detects potential divide-by-zero issues |
| Cppcheck | 2.11 | warning:possibleDivision | Checks for dangerous arithmetic operations |
| Coverity | 2024.06 | DIVIDE\_BY\_ZERO | Flags risky division operations |
| SonarQube | 9.9 | cpp:S3518 | Identifies unvalidated inputs |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-STR-001 | Ensure Safe and Correct Use of C-Style Strings. C-style strings are error-prone because they don’t include built-in bounds checking. If you’re not careful, you can easily overrun buffers or misuse functions like strcpy or strcat, which don’t stop you from writing past the end of memory. This can lead to crashes or security issues like buffer overflows. Using safer alternatives or being strict with length checks helps prevent these problems and keeps code more stable and secure. |

| **Noncompliant Code** |
| --- |
| This code doesn’t check the size of the destination buffer when copying a string, which could easily overflow and corrupt memory. |
| void copyInput(const char\* input) {  char buffer[16];  strcpy(buffer, input); // No size check  } |

| **Compliant Code** |
| --- |
| Here, we use strncpy, which limits how many characters can be copied to avoid going past the buffer’s size. It’s not perfect, but it’s safer than strcpy. |
| void copyInput(const char\* input) {  char buffer[16];  strncpy(buffer, input, sizeof(buffer) - 1);  buffer[15] = '\0'; // Make sure it’s null-terminated  } |

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

| **Principles(s):** Prevent Buffer Overflows. This principle applies because properly managing string length and boundaries stops attackers from writing beyond allocated memory and causing crashes or arbitrary code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.3 | BUFFER\_OVERFLOW | Detects buffer overflows and unsafe string ops |
| SonarQube | 9.9 LTS | c:S2183 | Flags unsafe usage of C-style string functions |
| Fortify | 23.1 | StringAnalyzer | Reviews string handling for potential risks |
| Cppcheck | 2.12 | - | Identifies unsafe strcpy/strncpy patterns |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-DBI | Preventing SQL Injection. SQL injection is a critical security issue that lets attackers manipulate a database by inserting malicious queries. Using parameterized queries and avoiding string concatenation helps stop these attacks and keeps data safe. |

| **Noncompliant Code** |
| --- |
| Builds the query by adding user input directly into the SQL string. |
| string username = getUserInput();  string query = "SELECT \* FROM users WHERE name = '" + username + "'";  db.execute(query); |

| **Compliant Code** |
| --- |
| Uses a parameterized statement to insert user input safely. |
| string username = getUserInput();  PreparedStatement stmt = db.prepare("SELECT \* FROM users WHERE name = ?");  stmt.bind(1, username);  stmt.execute(); |

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

| **Principles(s):** Limit Data Exposure. This principle applies because restricting direct access to database queries and filtering user input helps avoid exposing sensitive information through injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S3649 | Detects unsafe SQL query construction |
| Cppcheck | 2.10 | - | Warns on unsafe string handling |
| Fortify | 22.1 | SQLInjection | Identifies SQL injection patterns |
| CodeQL | Latest | Sql-injection | Flags vulnerable SQL usages |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-MEM | Safe Use of Memory. Improper memory management can lead to crashes, data corruption, or attacks like buffer overflows. Always initialize memory and avoid unsafe pointer use. |

| **Noncompliant Code** |
| --- |
| Allocates memory but does not initialize or validate usage. |
| int\* buffer = new int[10];  buffer[10] = 42; // out of bounds |

| **Compliant Code** |
| --- |
| [Initializes memory and ensures bounds are respected. |
| int\* buffer = new int[10]();  if (index >= 0 && index < 10) {  buffer[index] = 42;  } |

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

| **Principles(s):** Protect Memory. This principle applies because controlling access to memory and preventing improper reads or writes defends against data corruption and system compromise. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | memcheck | Checks for memory misuse |
| Clang Tidy | 16.0 | cppcoreguidelines | Warns about unsafe memory handling |
| Fortify | 22.1 | BufferOverflow | Detects buffer overflow patterns |
| Coverity | 2024.1 | ARRAY\_INDEX | Identifies out-of-bounds issues |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-ASN | Proper Use of Assertions. Assertions help catch logic errors early during testing. But they should not be used to enforce runtime constraints or validate user input. |

| **Noncompliant Code** |
| --- |
| Uses assertions to validate user input. |
| int age = getInput();  assert(age > 0); // not safe for release builds |

| **Compliant Code** |
| --- |
| Uses runtime checks and handles errors properly. |
| int age = getInput();  if (age <= 0) {  throw invalid\_argument("Age must be positive");  } |

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

| **Principles(s):** Fail Securely. This principle applies because ensuring that a program responds safely when conditions are not met helps avoid revealing system details or allowing unsafe operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | - | Flags bad assert usage |
| Clang Tidy | 16.0 | misc-assert | Identifies misuse of assert |
| CodeSonar | 8.4 | ASSERT\_ERROR | Finds improper assertions |
| Fortify | 22.1 | AssertUse | Warns about dangerous assert statements |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-EXC | Secure Exception Handling. Uncaught exceptions can crash the program or leave it in an unstable state. Always catch exceptions and avoid using them for regular control flow. |

| **Noncompliant Code** |
| --- |
| Allows an exception to propagate without handling. |
| void loadFile(string path) {  ifstream file(path);  if (!file) throw runtime\_error("File error");  } |

| **Compliant Code** |
| --- |
| Uses try-catch to handle errors gracefully. |
| void loadFile(string path) {  try {  ifstream file(path);  if (!file) throw runtime\_error("File error");  } catch (const exception& e) {  logError(e.what());  }  } |

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

| **Principles(s):** Handle Errors Properly. This principle applies because managing exceptions in a controlled way prevents crashes, undefined behavior, or exposure of sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Tidy | 16.0 | cert-err09-cpp | Catches bad exception usage |
| Cppcheck | 2.10 | - | Warns about uncaught exceptions |
| CodeSonar | 8.4 | EXCEPTIONS | Checks for missing or unsafe catch |
| Fortify | 22.1 | ErrorHandling | Identifies insecure error handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-008-INP | Validate All Input. Accepting unchecked input is dangerous. Validate all user input to ensure it meets expected formats and limits. |

| **Noncompliant Code** |
| --- |
| Accepts unchecked user input. |
| string name;  cin >> name; // no checks |

| **Compliant Code** |
| --- |
| Validates that input is alphabetic and length is safe. |
| string name;  cin >> name;  if (!regex\_match(name, regex("^[A-Za-z]{1,50}$"))) {  throw invalid\_argument("Invalid name input");  } |

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

| **Principles(s):** Least Privilege. This principle applies because granting only the minimum access necessary reduces the risk of unauthorized actions or privilege escalation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.1 | InputValidation | Flags lack of input validation |
| CodeQL | Latest | unsanitized-input | Identifies unsafe inputs |
| SonarQube | 9.9 | S2076 | Detects unchecked user input |
| Cppcheck | 2.10 | - | Warns about missing input validation |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Access | STD-009-FIO | Secure File Access. Files should not be opened without verifying paths or permissions. Failing to check file operations could lead to data leaks or corruption. |

| **Noncompliant Code** |
| --- |
| Opens a file directly from user input. |
| ifstream file(userPath); // risky |

| **Compliant Code** |
| --- |
| Checks file path and access rights before use. |
| if (isSafePath(userPath)) {  ifstream file(userPath);  } |

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

| **Principles(s):** Enforce Proper Data Ranges. This principle applies because ensuring that values are checked before use prevents security flaws that could arise from out-of-range or invalid data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.1 | FileAccess | Checks for unsafe file operations |
| Clang Tidy | 16.0 | misc-fileio | Warns on risky file handling |
| Cppcheck | 2.10 | - | Flags file access with user input |
| CodeSonar | 8.4 | FIO | Detects file I/O security problems |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging and Auditing | STD-010-LOG | Secure Logging Practices. Logs are helpful, but leaking sensitive info like passwords or personal data is dangerous. Always sanitize logs and protect them from unauthorized access. |

| **Noncompliant Code** |
| --- |
| Logs sensitive user data directly. |
| log("User password: " + password); |

| **Compliant Code** |
| --- |
| Masks sensitive data in logs. |
| log("User login attempt with masked password"); |

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

| **Principles(s):** Protect Memory. This principle applies because safeguarding memory allocation and access reduces the chance of corruption, leakage, or exploitation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Mdeium | High | Low | Moderate | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S2228 | Flags sensitive data in logs |
| Fortify | 22.1 | Logging | Detects insecure log content |
| CodeQL | Latest | sensitive-log | Warns on logging sensitive data |
| Cppcheck | 2.10 | - | Basic log misuse warnings |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation will be applied across every stage of the DevSecOps workflow to ensure that the secure coding standards in this policy are followed without interruption. In the pre-production cycle, automated processes such as risk evaluations, static code scans, and compliance validations will run during the “Assess and plan,” “Design,” and “Build” phases to detect and resolve potential weaknesses before they progress further. The “Verify and test” stage will rely on automated vulnerability scans and functional testing routines to confirm that each build meets security and quality expectations before it moves into production.

Once in production, automation will support real-time monitoring, detection, and alerting within the “Monitor and detect” phase, while scripted, predefined actions will allow for quick responses during the “Respond” stage. The “Transition and health check” phase will make use of automated configuration checks and security setting reviews to maintain consistent protection across systems. In the “Maintain and stabilize” phase, automation will handle baseline comparisons, stability assessments, and rollback operations, helping restore systems promptly after any disruption while keeping them aligned with policy requirements.

### 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 | High | Possible | Medium | High | 2 |
| STD-003-CPP | Medium | Possible | Low | Medium | 3 |
| STD-004-CPP | High | Likely | High | High | 1 |
| STD-005-CPP | High | Possible | Medium | High | 2 |
| STD-006-CPP | Medium | Unlikely | Medium | Medium | 3 |
| STD-007-CPP | Medium | Possible | Low | Medium | 3 |
| STD-008-CPP | High | Possible | Medium | High | 2 |
| STD-009-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-0010-CPP | Medium | Possible | Low | Medium | 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 secures stored information by converting it into a format that cannot be understood without the correct decryption key. This includes data on servers, local drives, and cloud platforms. The policy applies to protect sensitive records from exposure if the storage media is lost, stolen, or accessed without permission. |
| Encryption in flight | Encryption in flight safeguards information as it moves across internal networks or the internet by using secure transmission protocols such as TLS or SSL. The policy applies to prevent eavesdropping and data interception while information is traveling between systems or users. |
| Encryption in use | Encryption in use keeps data protected while it is actively being accessed, processed, or stored temporarily in system memory. The policy applies to defend against attacks that attempt to capture information during live operations, such as memory scraping or unauthorized runtime access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication confirms the identity of a user, device, or process before allowing entry into a system or application. The policy applies to ensure that only verified and trusted entities gain access, lowering the chance of unauthorized entry. |
| Authorization | Authorization controls the level of access granted to a verified user, defining which data, tools, or operations they are allowed to use. The policy applies to enforce security boundaries and prevent users from performing actions outside of their designated role. |
| Accounting | Accounting records and monitors user actions within a system, including logins, resource usage, and data changes. The policy applies to provide traceability, support incident investigations, and maintain compliance with organizational and legal requirements. |

**\***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

| **Rule** | **Principles** | **Justification** |
| --- | --- | --- |
| STD-001-CPP | 1, 3 | Principle 1 focuses on thorough input validation, ensuring only safe and expected values are processed. Principle 3 supports resilient error handling that prevents sensitive system details from being exposed during failures. |
| STD-002-CPP | 2, 4 | Principle 2 promotes careful management of system resources to avoid misuse or corruption. Principle 4 limits access strictly to what is necessary, reducing the potential for unauthorized activity. |
| STD-003-CPP | 5,7 | Principle 5 safeguards data in storage by applying strong protection methods. Principle 7 adds extra layers of defense so that if one control fails, others still maintain security. |
| STD-004-CPP | 1, 6 | Principle 1 enforces validation of all incoming data before it triggers important operations. Principle 6 emphasizes the use of secure development techniques to reduce vulnerabilities during coding. |
| STD-005-CPP | 2, 8 | Principle 2 ensures resources are properly handled from allocation to release to prevent security gaps. Principle 8 encourages ongoing oversight to detect unusual or risky behavior quickly. |
| STD-006-CPP | 3, 9 | Principle 3 ensures the system can handle errors gracefully while maintaining stability. Principle 9 requires comprehensive logging and auditing so that incidents can be investigated effectively. |
| STD-007-CPP | 4, 10 | Principle 4 applies strict access rules to prevent unauthorized entry. Principle 10 focuses on regularly reviewing and enhancing controls to keep them strong against new threats. |
| STD-008-CPP | 1, 7 | Principle 1 verifies that all data meets required conditions before being processed. Principle 7 relies on multiple security measures working together to protect against failures in a single control. |
| STD-009-CPP | 5, 8 | Principle 5 protects stored data from being accessed or altered without permission. Principle 8 ensures that activity is continuously monitored so suspicious patterns can be addressed quickly. |
| STD-0010-CPP | 6, 9 | Principle 6 promotes writing code with built-in safeguards that minimize weaknesses during execution. Principle 9 keeps detailed records of system activity, making it easier to confirm compliance and track issues. |

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 | 07/15/2025 | Added definitions for all 10 core security principles, included compliant and noncompliant code examples, and provided detailed explanations per Module 3 requirements | Bruce Gaudet | Mimi Tam |
| 1.2 | 08/08/2025 | Finalized document with justifications and completed Policy Version History table | Bruce Gaudet | Mimi Tam |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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