**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 12](#_Toc52464067)

[Coding Standard 10 13](#_Toc52464068)

[Defense-in-Depth Illustration 14](#_Toc52464069)

[Project One 14](#_Toc52464070)

[1. Revise the C/C++ Standards 14](#_Toc52464071)

[2. Risk Assessment 14](#_Toc52464072)

[3. Automated Detection 14](#_Toc52464073)

[4. Automation 14](#_Toc52464074)

[5. Summary of Risk Assessments 15](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 15](#_Toc52464076)

[7. Map the Principles 16](#_Toc52464077)

[Audit Controls and Management 17](#_Toc52464078)

[Enforcement 17](#_Toc52464079)

[Exceptions Process 17](#_Toc52464080)

[Distribution 18](#_Toc52464081)

[Policy Change Control 18](#_Toc52464082)

[Policy Version History 18](#_Toc52464083)

[Appendix A Lookups 18](#_Toc52464084)

[Approved C/C++ Language Acronyms 18](#_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 | Always check and sanitize user input to prevent unexpected or harmful data from entering the system. This helps prevent attacks like SQL injection and buffer overflows. |
| 1. Heed Compiler Warnings | Pay attention to warnings during compilation, as they often reveal potential vulnerabilities or bad coding practices that could lead to bugs or exploits. |
| 1. Architect and Design for Security Policies | Plan security from the start by designing your system to enforce policies like access control, authentication, and encryption. Security should be built into the architecture. |
| 1. Keep It Simple | Avoid overly complex code or architectures. Simple code is easier to understand, test, and secure, reducing the chances of introducing hidden vulnerabilities. |
| 1. Default Deny | By default, access to resources or functionalities should be denied unless explicitly allowed. This limits the impact of unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Each process or user should operate with the minimum permissions necessary. This limits damage if the system is compromised. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to other applications, databases, or services, ensure it’s properly validated and escaped to prevent injection or corruption. |
| 1. Practice Defense in Depth | Use multiple layers of security—such as firewalls, encryption, and access controls—so if one layer is breached, others remain in place to protect the system. |
| 1. Use Effective Quality Assurance Techniques | Include secure code reviews, static code analysis, and rigorous testing in the development cycle to catch vulnerabilities early. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards (like SEI CERT C++) to ensure consistent, secure development practices across all projects. |

### 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 | Using the correct data types in C++ helps protect your program from bugs, memory issues, or unexpected behavior. For example, using the wrong type can lead to data overflow or loss of precision. This standard keeps data safe and your program stable. |

| **Noncompliant Code** |
| --- |
| This code uses a char type for storing an age, which can only hold small numbers and could overflow. |
| char age = 130; // Too high for a char; causes overflow |

| **Compliant Code** |
| --- |
| This version uses an unsigned int, which is safer for storing positive whole numbers like age. |
| unsigned int age = 130; |

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

| **Principles(s):**  **Principle 2: Heed Compiler Warnings** – Compiler warnings often reveal issues with type mismatches that could cause security vulnerabilities or data loss.  **Principle 10: Adopt a Secure Coding Standard** – Enforces consistent data type usage throughout the application to reduce logic errors and overflow risks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium – Refactoring data types can affect other parts of the system. | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.17.1 | Type mismatch checker | Scans for incorrect type assignments and improper conversions. |
| Clang-Tidy | 20.0.0 | cppcoreguidelines‑pro‑type‑member‑init | Ensures class/struct members are initialized with proper types |
| PVS‑Studio | 7.25.49925 | Type error diagnostics | Catches type-dependent logic errors across modules |
| CppDepend | 2025.1.0 | CQLinq rules for type consistency | Ensures declared types match usage patterns and interfaces |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | This standard ensures that the values users enter are within safe, acceptable ranges. Even if the data type is correct, using values outside of expected limits can still crash the program or cause logic failures. |

| **Noncompliant Code** |
| --- |
| There is no check to make sure user Input is within a safe range, which could lead to unexpected behavior or crashes. |
| int temp = userInput; |

| **Compliant Code** |
| --- |
| This code safely checks the input before assigning it to a variable. |
| if (userInput >= 0 && userInput <= 100) {  int temp = userInput;  } |

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

| **Principles(s):**  **Principle 1: Validate Input Data** – Ensures values fall within expected safe ranges.  **Principle 9: Use Effective QA Techniques** – Promotes testing for value boundary conditions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | Value constraint analyzer | Detects anomalies in variable ranges |
| Cppcheck | 2.17.1 | Value range check | Flags variables that may exceed safe limits |
| PVS‑Studio | 7.25.49925 | Numeric overflow analysis | Catches possible integer overflow and underflow. |
| Klocwork | 2025.1.0 | Data-flow value analysis | Highlights unsafe operations based on execution paths |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Strings in C++ can be tricky. If you copy a string into a buffer that’s too small, it can overflow and corrupt memory. This standard is about safely managing string sizes and copying. |

| **Noncompliant Code** |
| --- |
| This code copies the full user input without checking if it fits, which can overflow the buffer. |
| char name[10];  strcpy(name, userInput); |

| **Compliant Code** |
| --- |
| The strncpy function limits how much is copied, preventing buffer overflow, and the last line ensures the string ends properly. |
| char name[10];  strncpy(name, userInput, sizeof(name) - 1);  name[9] = '\0'; // Ensure null-termination |

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

| **Principles(s):**  **Principle 7: Sanitize Data Sent to Other Systems** – Prevents string-based attacks and buffer overflows.  **Principle 10: Adopt a Secure Coding Standard** – Encourages safe string handling patterns. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2025.3.0 | BUFFER\_SIZE/string misuse | Finds buffer overflow risks in string functions. |
| Cppcheck | 2.17.1 | Buffer overflow detection | Flags unsafe functions like strcpy, sprintf. |
| PVS‑Studio | 7.25.49925 | Buffer overrun checks | Detects off-by-one and underrun errors with strings. |
| Polyspace | R2022b | String safety cops | Verifies compliance with CERT C++ string practices |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Directly inserting user input into a SQL statement is dangerous. It allows attackers to change the query and steal or damage data. This standard promotes the use of safe, parameterized queries. |

| **Noncompliant Code** |
| --- |
| This code builds a query using user input, which can allow attackers to inject malicious SQL code. |
| string query = "SELECT \* FROM users WHERE name='" + user + "'"; |

| **Compliant Code** |
| --- |
| Using prepared statements ensures the input is handled safely and not treated as part of the SQL code. |
| stmt->prepare("SELECT \* FROM users WHERE name=?");  stmt->bind(user); |

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

| **Principles(s):**  **Principle 1: Validate Input Data** – Disallows untrusted data in queries.  **Principle 7: Sanitize Data Sent to Other Systems** – Prevents SQL injection threats. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | SQL Injection detection | Flags unsafe string concatenation in queries |
| Fortify SCA | 24.4.0 | SQL/Command injection rules | Identifies unparameterized SQL patterns . |
| PVS‑Studio | 7.25.49925 | SQL injection path analysis | Traces tainted data into SQL executions. |
| CppDepend | 2025.1.0 | Custom CQLinq query rules | Enables project-specific detection of unsafe query practices. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | After you delete a pointer, trying to use it again is very risky. It could lead to crashes or allow attackers to mess with memory. This standard makes sure memory is used safely. |

| **Noncompliant Code** |
| --- |
| This code uses a pointer after deleting it, which is unsafe and unpredictable. |
| int\* ptr = new int;  delete ptr;  \*ptr = 5; // Use after delete |

| **Compliant Code** |
| --- |
| After deleting the pointer, we reset it to nullptr, which prevents it from being used by accident. |
| int\* ptr = new int;  delete ptr;  ptr = nullptr; |

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

| **Principles(s):**  **Principle 6: Least Privilege** – Restricts unintended memory access.  **Principle 8: Defense in Depth** – Adds multiple layers of memory safety. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind (Memcheck) | 3.25.0 | Memory access, leaks | Detects invalid read/write and unfreed allocations |
| AddressSanitizer (Clang/GCC) | 1.0.0 | Runtime memory errors | Detects buffer overflows and use-after-free issues |
| Intel Inspector | 2021.4 | Memory/thread debugger | Detects memory corruption, thread-related misuse |
| Insure++ | 2021.2 | Source-level memory instrumentation | Detects leaks, use‑after‑free, double freeing |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions help catch mistakes while you're developing, but they shouldn't replace proper checks in real programs. This standard reminds us to write real safety checks, not just use assert. |

| **Noncompliant Code** |
| --- |
| This only checks in debug mode. If ptr is null in a release build, it will crash. |
| assert(ptr != nullptr);  \*ptr = 10; |

| **Compliant Code** |
| --- |
| The code safely checks if the pointer is usable before trying to use it. |
| if (ptr != nullptr) {  \*ptr = 10;  } |

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

| **Principles(s):**  **Principle 4: Keep It Simple** – Avoids complexity masked by assertions.  **Principle 9: Use Effective QA Techniques** – Promotes legitimate runtime validations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | Assertion misuse/unreachable code | Flags assertions in production code |
| GoogleTest | (current stable) | Unit testing framework | Supports test assertions and coverage validation. |
| PVS‑Studio | 7.25.49925 | Test assertion patterns | Detects redundant or misused assertions. |
| SonarQube | 10.4 | Assertion and test code review | Highlights ineffective or missing assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Throwing exceptions is a good way to handle errors, but you should always provide a clear message or type. This helps others know what went wrong and how to fix it. |

| **Noncompliant Code** |
| --- |
| This throws an exception without giving a message, which makes it hard to understand. |
| throw; |

| **Compliant Code** |
| --- |
| This throws a clear error message, which helps with debugging and logging. |
| throw std::runtime\_error("Invalid operation"); |

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

| **Principles(s):**  **Principle 3: Design for Security Policies** – Guarantees predictable failure modes.  **Principle 9: Use Effective QA Techniques** – Fosters robust exception handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang‑Tidy | 20.0.0 | misc-throw-by-value-catch-by-reference | Detects exception handling anti‑patterns. |
| SonarQube | 10.4 | Exception handling rules | Flags empty catches or catch‑all blocks |
| PVS‑Studio | 7.25.49925 | Exception safety compliance | Detects leaks and unsafe exception flow. |
| CppDepend | 2025.1.0 | CQLinq exception use rules | Enforces consistent exception design patterns. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Ownership | STD-008-CPP | Using raw pointers without clear ownership can lead to memory leaks or double deletes. This standard encourages using smart pointers, which automatically manage memory. |

| **Noncompliant Code** |
| --- |
| The pointer is never deleted, so the memory is never released. |
| int\* ptr = new int(5); // No delete |

| **Compliant Code** |
| --- |
| Smart pointers delete memory automatically when they're done being used. |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(5); |

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

| **Principles(s):**  **Principle 6: Least Privilege** – Limits pointer misuse and memory exposure.  **Principle 10: Adopt a Secure Coding Standard** – Promotes use of std::unique\_ptr, std::shared\_ptr |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | LLVM 21.x | Memory leak/path misuse | Detects unowned/dangling pointers. |
| Cppcheck | 2.17.1 | Pointer cleanup issues | Flags mismatched free/delete or unused pointers |
| PVS‑Studio | 7.25.49925 | Smart pointer misuse detection | Discovers raw pointer ownership issues. |
| CppDepend | 2025.1.0 | Ownership rule sets via CQLinq | Enforces use of RAII patterns and smart pointers. |

Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Initialization | STD-009-CPP | Variables must be given a value before you use them. If you don’t, they might hold garbage data that breaks your program. |

| **Noncompliant Code** |
| --- |
| Using x before setting a value can cause random or bad results. |
| int x;  int y = x + 1; // x is uninitialized |

| **Compliant Code** |
| --- |
| Here, we initialize x before using it, which avoids errors. |
| int x = 0;  int y = x + 1; |

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

| **Principles(s):**  **Principle 2: Heed Compiler Warnings** – Catches use-before-init at compile time.  **Principle 10: Adopt a Secure Coding Standard** – Encourages secure defaults. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC / Clang | latest | -Wuninitialized | Warns on use-before-init during compile time. |
| Cppcheck | 2.17.1 | Uninitialized variable alerts | Flags variables that lack default values |
| PVS‑Studio | 7.25.49925 | Initialization protection analysis | Detects missing constructors or default values. |
| Coverity | 2025.3.0 | Uninitialized variable detection | Finds initialization issues in complex paths. |

Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Boundary Checks | STD-010-CPP | If you try to access part of an array that doesn’t exist, your program might crash or even leak data. This standard requires checking that your index is within bounds. |

| **Noncompliant Code** |
| --- |
| This code writes past the end of the array, which is dangerous and unpredictable. |
| int arr[5];  arr[10] = 3; // Out of bounds |

| **Compliant Code** |
| --- |
| [Compliant description] |
| if (i >= 0 && i < 5) {  arr[i] = 3;  } |

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

| **Principles(s):**  **Principle 1:** Validate Input Data  **Principle 8: Defense in Depth** – Ensures array/pointer bounds checks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2025.3.0 | ARRAY\_INDEX\_OUT\_OF\_BOUNDS | Flags out-of-range array accesses |
| AddressSanitizer (Clang/GCC) | 1.0.0 | Runtime buffer overflow detection | Detects invalid memory access when executing. |
| PVS‑Studio | 7.25.49925 | Array bounds analysis | Identifies indexing logic errors. |
| Valgrind | 3.25.0 | Accessing invalid memory | Detects out-of-range reads/writes |

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.

Green Pace integrates security automation into every phase of the software development lifecycle to ensure continuous compliance and protection. During the pre-production phase, automation begins with the “Assess and Plan” step, where tools like threat modeling platforms and static code analyzers are used to evaluate vulnerabilities early. In the Design and Build stages, secure coding tools and dependency scanning are automated within IDEs and CI/CD pipelines to prevent insecure libraries and logic flaws from being introduced.

As the team moves into Verify and Test, security testing is automated through fuzzers, unit test frameworks, and tools like Clang Sanitizers or AddressSanitizer to identify memory issues, boundary violations, or input errors. This step ensures only secure, verified code transitions into production.

In the Production phase, automation focuses on Transition and Health Check via penetration testing and baseline checks using tools like Fortify SCA or Coverity. Once deployed, Monitor and Detect uses SIEM platforms like Splunk or Azure Sentinel to automate log collection, anomaly detection, and alerting based on behavior analytics. In Respond and Maintain and Stabilize, response orchestration tools like SOAR platforms automate threat response, rollbacks, and restoration to a secure baseline.

Overall, this pipeline supports the enforcement of Green Pace’s secure coding standards while reducing human error and increasing efficiency in detecting and responding to vulnerabilities.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Likely | Low | Medium | 3 |
| STD-003-CPP | High | Likely | High | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Possible | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | Medium | Likely | Low | Medium | 3 |
| STD-010-CPP | High | Likely | Medium | High | 5 |

### 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 means protecting your data when it’s stored on a hard drive, USB, or in the cloud. At Green Pace, all stored files like customer information or saved reports must be locked using strong encryption, such as AES-256. This helps protect the data in case someone steals a computer or accesses the storage without permission. |
| Encryption in flight | Encryption in flight is how we protect data while it's being sent from one place to another like when you're logging into a website or sending files over the internet. Green Pace uses secure connections like TLS 1.3 (kind of like HTTPS) to make sure no one can spy on that information while it’s moving between systems, apps, or people. |
| Encryption in use | Encryption in use means protecting data even while it’s being worked on like when a program is using it in memory (RAM). At Green Pace, we use special tools like memory encryption or secure environments to make sure that data is safe even while it’s being processed or analyzed. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication means checking if someone really is who they say they are. At Green Pace, we use multi-factor authentication (MFA). That means you need more than just a password like a phone app or a key code to log in. We do this for all important systems, like emails, cloud tools, or company servers. |
| Authorization | Authorization decides what someone can do once they’re logged in. Green Pace follows the “least privilege” rule which means you only get access to what you need for your job. For example, a developer might be able to view system logs but can’t change important files. This helps keep systems safe and limits accidents. |
| Accounting | Accounting means recording what everyone does in the system. Every time someone logs in, opens a file, or changes something important, it’s saved in a log. Green Pace uses log tracking tools to keep records of everything, in case we need to review what happened or investigate a problem later. |

**\***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

**Mapping Security Principles to Coding Standards**

1. STD-001-CPP – Data Type
   1. 2. Heed Compiler Warnings – Using the wrong data type often causes compiler warnings. Paying attention to those helps avoid bugs.
   2. 10. Adopt a Secure Coding Standard – Using the right data type is part of writing safe and consistent code.
2. STD-002-CPP – Data Value
   1. 1. Validate Input Data – You need to make sure values are safe and not too big or too small.
   2. 9. Use Effective Quality Assurance Techniques – Testing can help catch bad values early.
3. STD-003-CPP – String Correctness
   1. 7. Sanitize Data Sent to Other Systems – Strings can break things or be used in attacks if you don’t clean them.
   2. 10. Adopt a Secure Coding Standard – Safe string functions are part of secure coding.
4. STD-004-CPP – SQL Injection
   1. 1. Validate Input Data – You must check user input so no one can sneak in bad code.
   2. 7. Sanitize Data Sent to Other Systems – SQL commands need to be protected from user input.
5. STD-005-CPP – Memory Protection
   1. 6. Adhere to the Principle of Least Privilege – If a program only uses memory it needs, it’s safer.
   2. 8. Practice Defense in Depth – Having memory safety adds another layer of protection.
6. STD-006-CPP – Assertions
   1. 4. Keep It Simple – Don’t rely too much on assertions that only work in test mode.
   2. 9. Use Effective Quality Assurance Techniques – Real testing is better than hoping an assertion catches problems.
7. STD-007-CPP – Exceptions
   1. 3. Architect and Design for Security Policies – Having a plan for what to do when things go wrong is part of a strong design.
   2. 9. Use Effective Quality Assurance Techniques – Good error handling helps during testing and debugging.
8. STD-008-CPP – Pointer Ownership
   1. 6. Adhere to the Principle of Least Privilege – Using smart pointers helps avoid giving too much control.
   2. 10. Adopt a Secure Coding Standard – Secure coding means managing memory carefully.
9. STD-009-CPP – Initialization
   1. 2. Heed Compiler Warnings – Compilers often warn when variables are not set.
   2. 10. Adopt a Secure Coding Standard – Always setting variables before use is part of writing good, safe code.
10. STD-010-CPP – Boundary Checks
    1. 1. Validate Input Data – You must check array indexes so they don’t go too far.
    2. 8. Practice Defense in Depth – Boundary checks help stop one small mistake from causing big problems.

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 | 05/10/2025 | Updated with Module 3 milestone: Added core security principles and draft coding standards | Darrell Walker | Instructor |
| 1.2 | 06/13/2025 | Finalized Project One: Added automation tools, encryption policies, AAA framework, risk assessments, and mapped principles to standards | Darrell Walker | Instructor |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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