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



# Green Pace Secure Development Policy

# Mohamed Aziz Zaghdoudi

## 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 | Input validation ensures that data received by the application is correct, safe, and expected. This involves validating format, type, range, and length to avoid buffer overflows, SQL injection, and other attacks that exploit unchecked inputs. |
| 1. Heed Compiler Warnings | Compiler warnings are essential indicators of potential coding issues. Addressing these warnings can prevent vulnerabilities such as memory leaks, data corruption, and undefined behavior that could be exploited by attackers. |
| 1. Architect and Design for Security Policies | Security should be integrated into the architecture and design phase. This includes defining security requirements, threat modeling, and employing security mechanisms that address identified risks during the design process. |
| 1. Keep It Simple | Simplicity in design and implementation reduces the likelihood of introducing security vulnerabilities. Simple code is easier to review, test, and maintain, making it less prone to flaws that attackers could exploit. |
| 1. Default Deny | Implementing a default deny policy ensures that access is explicitly granted rather than assumed. By default, all access should be denied unless specific permissions are provided, minimizing the risk of unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | This principle involves granting only the necessary permissions required for a specific task or user. Limiting privileges reduces the attack surface and minimizes the potential damage from a security breach. |
| 1. Sanitize Data Sent to Other Systems | Data that is transmitted to external systems must be properly sanitized to prevent injection attacks, corruption, or misuse. This includes filtering, encoding, or escaping data as required. |
| 1. Practice Defense in Depth | Using multiple layers of security controls provides redundancy in case a single layer is compromised. This approach involves combining preventive, detective, and responsive. |
| 1. Use Effective Quality Assurance Techniques | Thorough testing, including code reviews, automated testing, and static analysis, helps identify vulnerabilities early in the development process. Quality assurance practices are essential for ensuring secure code. |
| 1. Adopt a Secure Coding Standard | Following established secure coding standards, such as SEI CERT C++ Coding Standard, ensures consistency and addresses common vulnerabilities. Compliance with standards helps prevent security flaws from being introduced during development. |

### 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 proper data types ensures type safety and avoids vulnerabilities like buffer overflows and integer overflows. |

| **Noncompliant Code** |
| --- |
| Incorrectly using a data type that cannot handle the intended value, leading to data loss or incorrect behavior. |
| int largeValue = 2147483648; // Error: Value exceeds int range |

| **Compliant Code** |
| --- |
| Using a suitable data type to handle large values without overflow issues. |
| long long largeValue = 2147483648LL; // Correct: Uses long long for large values |

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

| **Principles(s):** Principle 1 (Validate Input Data), Principle 2 (Heed Compiler Warnings), Using the correct data type avoids incorrect memory usage and unsafe input assignments. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | readability-implicit-bool-cast | Detects mismatches between data types and values |
| Clang-Tidy | 14.0 | readability-implicit-bool-cast | Detects incorrect implicit type conversions |
| Coverity | 2023.6 | OVERRUN\_STATIC | Detects integer overflows and memory overuse |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating data values prevents data corruption and security breaches by ensuring data meets expected criteria. |

| **Noncompliant Code** |
| --- |
| Accepting user input without validating if it is within an acceptable range, which could lead to unexpected behavior. |
| int age;  std::cin >> age; // No validation applied |

| **Compliant Code** |
| --- |
| Validating user input to ensure it meets acceptable criteria before further processing. |
| int age;  std::cin >> age;  if (age >= 0 && age <= 120) {  std::cout << "Valid age.";  } else {  std::cout << "Invalid age.";  } |

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

| **Principles(s):** Principle 1 (Validate Input Data), Principle 9 (Use Effective Quality Assurance Techniques), Ensures data is constrained to expected safe values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S3518 | Detects missing input validation |
| Cppcheck | 2.12 | uninitVar | Warns on uninitialized/misused inputs |
| Clang-Tidy | 14.0 | bugprone-integer-division | Detects risky numerical operations |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Proper string handling prevents vulnerabilities like buffer overflows and string manipulation attacks. |

| **Noncompliant Code** |
| --- |
| Using unsafe functions that do not check string boundaries, which may result in buffer overflows. |
| char buffer[10];  strcpy(buffer, "This is a long string"); |

| **Compliant Code** |
| --- |
| Using safer string handling functions that enforce buffer size limits to prevent overflows. |
| char buffer[10];  strncpy(buffer, "Short", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; |

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

| **Principles(s):** Principle 4 (Keep It Simple), Principle 7 (Sanitize Data Sent to Other Systems) — Avoids buffer overflows through safe and limited string copying. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.2 | Buffer Overflow | Detects unsafe functions like strcpy |
| Cppcheck | 2.12 | bufferAccessOutOfBounds | Identifies potential buffer overflow risks |
| Clang-Tidy | 14.0 | cert-str34-c | Enforces bounds-checking on string operations |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Preventing SQL injection attacks by properly sanitizing inputs used in SQL queries. |

| **Noncompliant Code** |
| --- |
| Constructing SQL queries directly with user input, which is prone to injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Using prepared statements to safely handle user input and avoid SQL injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement stmt = conn.prepareStatement(query);  stmt.setString(1, userInput); |

**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), Principle 3 (Architect and Design for Security Policies) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S3649 | Flags direct SQL injection patterns |
| Fortify | 22.2 | SQL Injection | Detects unescaped or raw SQL inputs |
| Coverity | 2023.6 | SQL\_INJECTION | Finds unsafe string concatenation in queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Ensuring proper memory allocation and deallocation prevents memory leaks and buffer overflow vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Failing to properly deallocate dynamically allocated memory, which can result in memory leaks and potentially exhaust system resources. |
| int\* arr = new int[10]; // Memory leak: No delete[] call |

| **Compliant Code** |
| --- |
| Using proper memory management techniques to ensure memory is correctly freed when it is no longer needed. |
| int\* arr = new int[10];  delete[] arr; // Correctly freeing memory |

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

| **Principles(s):** Principle 6 (Least Privilege), Principle 9 (Quality Assurance Techniques) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | memcheck | Detects memory leaks, invalid frees |
| Cppcheck | 2.12 | memleak, unmatchedAlloc | Warns about memory not freed |
| Clang-Tidy | 14.0 | cppcoreguidelines-owning-memory | Detects improper memory ownership |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Proper use of assertions ensures program logic integrity and prevents unintended behavior. |

| **Noncompliant Code** |
| --- |
| Using assertions for input validation rather than for detecting programming errors, which is inappropriate for runtime checks. |
| assert(userInput > 0); // Using assertion for runtime error checking - Incorrect |

| **Compliant Code** |
| --- |
| Replacing assertions with condition checking to properly handle input validation at runtime. |
| if (userInput > 0) {  // Valid input, proceed  } else {  std::cerr << "Invalid input.";  } |

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

| **Principles(s):** Principle 5 (Default Deny), Principle 9 (Quality Assurance Techniques) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | cert-err33-c | Ensures assertions are not misused |
| Cppcheck | 2.12 | assertWithSideEffect | Detects invalid or misused assertions |
| SonarQube | 9.9 | cpp:S3518 | Highlights unsafe logic checks in production |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handling exceptions appropriately prevents application crashes and maintains program robustness. |

| **Noncompliant Code** |
| --- |
| Failing to check for potential errors or exceptions, leading to application crashes or undefined behavior. |
| int divide(int a, int b) {  return a / b; // No handling for division by zero  } |

| **Compliant Code** |
| --- |
| Using exception handling to manage potential runtime errors and ensure program stability. |
| int divide(int a, int b) {  try {  if (b == 0) throw std::runtime\_error("Division by zero");  return a / b;  } catch (const std::runtime\_error& e) {  std::cerr << e.what() << std::endl;  return 0;  }  } |

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

| **Principles(s):** Principle 4 (Keep It Simple), Principle 9 (Quality Assurance Techniques) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | misc-throw-by-value-catch-by-reference | Enforces safe exception usage |
| Coverity | 2023.6 | UNCAUGHT\_EXCEPT | Detects unhandled exceptions |
| SonarQube | 9.9 | cpp:S112 | Flags general catch without error handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | [STD-008-CPP] | Ensuring files are opened, read, written, and closed properly to avoid unauthorized access or data loss. |

| **Noncompliant Code** |
| --- |
| Failing to check if the file was successfully opened, which can lead to unexpected behavior or data corruption. |
| std::ofstream file("important.txt");  file << "Sensitive data.";  file.close(); |

| **Compliant Code** |
| --- |
| Checking the file state before performing operations and ensuring safe file handling. |
| std::ofstream file;  file.open("important.txt", std::ios::out | std::ios::binary);  if (file.is\_open()) {  file << "Sensitive data.";  file.close();  } else {  std::cerr << "Failed to open file.";  } |

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

| **Principles(s):** Principle 6 (Least Privilege), Principle 7 (Sanitize Data Sent to Other Systems) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.2 | File Access | Detects improper file operations |
| SonarQube | 9.9 | cpp:S2676 | Flags unsafe file access without validation |
| Cppcheck | 2.12 | fileAccess | Detects insecure or unchecked file handling |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Mgmt | [STD-009-CPP] | Ensuring resources are properly allocated and deallocated to prevent leaks and corruption. |

| **Noncompliant Code** |
| --- |
| Using smart pointers to automatically manage resource deallocation. |
| int\* ptr = new int[10]; // No delete[] call, causing memory leak |

| **Compliant Code** |
| --- |
| Using smart pointers to automatically manage resource deallocation. |
| std::unique\_ptr<int[]> ptr(new int[10]); // Uses smart pointer for automatic cleanup |

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

| **Principles(s):** Principle 6 (Least Privilege), Principle 8 (Defense in Depth) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | memcheck | Detects memory/resource leaks |
| Clang-Tidy | 14.0 | modernize-use-smart-ptr | Recommends use of smart pointers |
| Cppcheck | 2.12 | resourceLeak | Detects lost resources (file, memory, handles) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Logging | [STD-010-CPP] | Logging errors properly helps in troubleshooting and security auditing. |

| **Noncompliant Code** |
| --- |
| Using standard output for logging errors, which is not secure or traceable. |
| std::cerr << "An error occurred."; |

| **Compliant Code** |
| --- |
| Using a log file to properly document errors for further analysis. |
| std::ofstream logFile("error.log", std::ios::app);  if (logFile.is\_open()) {  logFile << "[ERROR] An error occurred." << std::endl;  logFile.close();  } |

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

| **Principles(s):** Principle 10 (Adopt a Secure Coding Standard), Principle 9 (Use Effective Quality Assurance Techniques) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S106 | Warns on improper logging practices |
| Fortify | 22.2 | Log Forging | Flags improper user-influenced logs |
| Clang-Tidy | 14.0 | misc-definitions-in-headers | Detects improper logging in global contexts |

### 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.

At Green Pace, automation isn’t just a convenience—it's essential to embedding security into every step of software development. Through the DevSecOps model, we bring together development, security, and operations teams to build a secure delivery pipeline that catches issues early, enforces policies consistently, and helps us move fast without compromising trust.

Here’s how automation supports secure coding practices across the lifecycle:

* Plan – Security starts at the beginning. During sprint planning, we identify potential risks and integrate them into the user stories. This proactive step ensures teams are aligned on security from day one.
* Develop – Developers work with secure code libraries and automated pre-commit hooks that flag risky patterns before the code even makes it into the repository.
* Build – Static analysis tools like Cppcheck and Clang-Tidy are baked into the build process. They scan for violations of coding standards and catch common vulnerabilities automatically.
* Test – Security tests are not an afterthought—they're part of our test suite. Automated testing includes fuzzing, input validation, and regression checks to catch potential flaws before they go live.
* Release – No release goes out without being scanned for known vulnerabilities. If something risky is found, the release is halted until resolved.
* Deploy – Access control, key management, and deployment validation checks are enforced automatically. This ensures environments are consistent and protected from misconfiguration.
* Operate – Once deployed, our systems log everything from API activity to access changes. These logs are actively monitored to detect anything unusual.
* Monitor – With integrated monitoring and SIEM (Security Information and Event Management) tools, we’re constantly watching for threats. If an anomaly is detected, the team is alerted in real time.

In short, automation helps us build security into the process, not bolt it on at the end. It reduces human error, enforces compliance, and gives us confidence that every change meets our high security standards before it’s ever deployed.

### 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 | Likely | Medium | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | High | Likely | Medium | High | 5 |
| STD-008-CPP | Medium | Likely | Low | Medium | 3 |
| STD-009-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-010-CPP | Medium | Likely | 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 | Protects stored data (e.g., databases, log files) using AES-256 or similar. Ensures that sensitive information is not readable if storage is compromised. |
| Encryption in flight | Applies to data in transit (e.g., API calls, web forms) using protocols like TLS 1.3. This prevents data interception or tampering during transmission. |
| Encryption in use | Protects data being processed in memory (e.g., through CPU-level isolation or encrypted memory). Useful in multi-tenant environments to protect active data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Confirms the identity of users or systems. Uses multi-factor authentication and secure credential management. Enforced during user login, API access. |
| Authorization | Grants or denies access to system features based on user roles. Enforced for database changes, file access, and administrative actions. |
| Accounting | Logs and monitors system activity for audit purposes. Tracks logins, file access, and database changes to support compliance and traceability. |

**\***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 | 04/10/2025 | Completed Project One revisions: coding standards, risk assessment, automation, encryption, and Triple-A policies | Mohamed Aziz Zaghdoudi |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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