**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview](#bookmark) 2

[Purpose](#bookmark1) 2

[Scope](#bookmark2) 2

[Module Three Milestone](#bookmark3) 2

[Ten Core Security Principles](#bookmark4) 2

[C/C++ Ten Coding Standards](#bookmark5) 3

[Coding Standard 1](#bookmark6) 4

[Coding Standard 2](#bookmark7) 5

[Coding Standard 3](#bookmark8) 6

[Coding Standard 4](#bookmark9) 7

[Coding Standard 5](#bookmark10) 8

[Coding Standard 6](#bookmark11) 9

[Coding Standard 7](#bookmark12) 10

[Coding Standard 8](#bookmark13) 11

[Coding Standard 9](#bookmark14) 13

[Coding Standard 10](#bookmark15) 14

[Defense-in-Depth Illustration](#bookmark16) 15

[Project One](#bookmark17) 15

[1. Revise the C/C++ Standards](#bookmark18) 15

[2. Risk Assessment](#bookmark19) 15

[3. Automated Detection](#bookmark20) 15

[4. Automation](#bookmark21) 15

[5. Summary of Risk Assessments](#bookmark22) 16

[6. Create Policies for Encryption and Triple A](#bookmark23) 16

[7. Map the Principles](#bookmark24) 17

[Audit Controls and Management](#bookmark25) 18

[Enforcement](#bookmark26) 18

[Exceptions Process](#bookmark27) 18

[Distribution](#bookmark28) 19

[Policy Change Control](#bookmark29) 19

[Policy Version History](#bookmark30) 19

[Appendix A Lookups](#bookmark31) 19

[Approved C/C++ Language Acronyms](#bookmark32) 19

## 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 must be validated to prevent malicious data from entering the system or causing unintended program behavior. Validation should include checks for data type, length, format and acceptable range. Proper input validation significantly reduces the risk of buffer overflows, injection attacks and other unexpected behaviors. |
| 1. Heed Compiler Warnings | Compiler warnings should always be taken seriously because they alert developers to unsafe coding practices, type mismatches or potential security flaws. Instead of suppressing these warnings, developers should resolve them to ensure their code runs more reliably and securely. |
| 1. Architect and Design for Security Policies | Security should be incorporated during the architecture and design phase not added later in development. Building with security in mind from the start ensures that access control, encryption and secure data handling are properly integrated into the system’s foundation. |
| 1. Keep It Simple | Overly complex code and systems increases the risk of vulnerabilities. Simple designs are easier to maintain and secure which reduces the chance of overlooked weaknesses. |
| 1. Default Deny | Systems and applications should deny access by default and only grant permissions that are required for functionality. This prevents unauthorized access and enforces stricter control over data and system resources. |
| 1. Adhere to the Principle of Least Privilege | Users should only have the minimum level of access necessary to perform their tasks. Enforcing least privilege reduces the potential damage from compromised accounts or processes. |
| 1. Sanitize Data Sent to Other Systems | Data shared across systems or components should be sanitized to ensure malicious content is not passed through. Proper sanitization protects against attacks such as cross-site scripting and injection vulnerabilities. |
| 1. Practice Defense in Depth | Multiple layers of security should be implemented such as authentication, encryption, firewalls and intrusion detection. This approach ensures that if one layer fails, others remain to protect the system. |
| 1. Use Effective Quality Assurance Techniques | It’s important to conduct thorough testing, perform code reviews and use analysis tools to identify security flaws early. Continuous quality assurance ensures code complies with standards and remains resilient against attacks. |
| 1. Adopt a Secure Coding Standard | Following established coding standards helps developers avoid common errors and reinforces best practices. Consistent standards make it easier to prevent and address vulnerabilities. |

### 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] | Appropriate data types should be selected to prevent overflow, truncation or other unintended type conversions. |

| **Noncompliant Code** |
| --- |
| This code assigns a value of 200 which exceeds the maximum range of the char data type. This causes an integer overflow, leading to undefined or unexpected behavior. |
| char age = 200; |

| **Compliant Code** |
| --- |
| This code uses the int data type which can safely store the value of 200. Selecting an appropriate data type prevents overflow and ensures the program works as intended. |
| int age = 200; |

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

|  |
| --- |
| **Principles(s):**   * Validate Input Data (Principle 1)   Using correct data types prevents overflow and logic errors.   * Keep It Simple (Principle 4)   Keeping types simple ensures readability and reduces hidden complexity. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.13 | invalidIntegerOverflow | This tool detects unsafe type conversions that can cause integer overflows and/or data loss. |
| Clang-Tidy | 17.0 | cppcoreguidelines-narrowing-conversions | This tool provides a warning when narrowing conversions occur that can lead to data truncation or unexpected behavior. |
| SonarQube | 10.5 | cpp:S1911 (Implicit casts) | This tool flags implicit casts that can lead to unpredictable results or data corruption. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Values should be validated before use especially when they come from external input. |

| **Noncompliant Code** |
| --- |
| This code uses an index entered by the user without validation. If the value of the variable index is negative or exceeds the bounds of the array, it can cause out-of-bounds access potentially leading to crashes or undefined behavior. |
| int index = userInput; array[index] = 10; |

| **Compliant Code** |
| --- |
| This code validates the value of the variable index before accessing the array. By ensuring the index is within valid bounds, it prevents out-of-bounds errors and enhances program safety. |
| if (index >= 0 && index < arraySize) {  array[index] = 10; } |

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

|  |
| --- |
| **Principles(s):**   * Validate Input Data (Principle 1)   Ensures that input values are within safe ranges.   * Default Deny (Principle 5)   Unsafe input should be denied by default. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17.0 | Core.OutOfBounds | This tool identifies null dereferences and out-of-bounds access. |
| Coverity | 2024.09 | TAINTED\_SCALAR | This tool flags tainted or unchecked user data. |
| SonarLint | 3.19 | cpp:S3510 (Validate User Input) | This tool warns when input is used without proper validation. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Use safe string handling to prevent buffer overflows and memory corruption. |

| **Noncompliant Code** |
| --- |
| This code copies a string into a fixed-size buffer without checking the string’s length. If the string exceeds the buffer size, it can cause a buffer overflow which can lead to crashes, undefined behavior or other security vulnerabilities. |
| char buffer[10]; strcpy(buffer, "This string is way too long!"); |

| **Compliant Code** |
| --- |
| This code uses strncpy to copy the string which limits the number of characters to the buffer size minus one, and explicitly null-terminates the buffer. This prevents buffer overflows and ensures safe string handling. |
| char buffer[10]; strncpy(buffer, "Safe", 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):**   * Sanitize Data Sent to Other Systems (Principle 7)   Ensures output data is sanitized before it is transmitted.   * Use Effective Quality Assurance Techniques (Principle 9)   Safe string handling avoids buffer overflows. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.13 | strcpyBufferOverflow | This tool identifies unsafe string functions that could cause buffer overflows. |
| SonarQube | 10.5 | cpp:S1199 (Buffer Overflow Risk) | This tool flags potential buffer overflow risks. |
| Fortify SCA | 24.1 | Buffer Overflow rulepack | This tool recommends safe alternatives like strcpy, memcpy\_s or std::string. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Always use parameterized queries or prepared statements to prevent SQL injection. |

| **Noncompliant Code** |
| --- |
| This code constructs a SQL query by concatenating user input. That approach is vulnerable to SQL injection, allowing attackers to manipulate the query and access or modify unauthorized data. |
| string query = "SELECT \* FROM users WHERE id = " + userInput; |

| **Compliant Code** |
| --- |
| This code uses prepared statements with parameter binding. By separating the SQL logic from user input, it prevents SQL injection attacks and ensures safe database access. |
| sqlite3\_prepare\_v2(db, "SELECT \* FROM users WHERE id = ?", -1, &stmt, 0); sqlite3\_bind\_int(stmt, 1, userId); |

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

|  |
| --- |
| **Principles(s):**   * Validate Input Data (Principle 1)   Parameterized queries enforce secure input validation making sure all input is checked and safe before use.   * Architect and Design for Security Policies (Principle 3)   Aligns with secure system design principles by building security into the design of the system.   * Default Deny (Principle 5)   System should deny unvalidated input by default. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | cpp:S3649 (SQL Injection) | This tool detects unsafe string concatenation in SQL queries |
| Checkmarx | 9.6 | SQL\_Injection | This tool identifies SQL query construction that could be exploited. |
| Fortify SCA | 24.1 | SQL Injection rule | This tool enforces the use of prepared statements or parameterized inputs to prevent injection. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Use smart pointers instead of raw pointers to prevent memory leaks and dangling references. |

| **Noncompliant Code** |
| --- |
| This code manually allocates memory and deletes it. After deletion, the pointer p becomes a dangling pointer which can lead to undefined behavior if accessed. |
| int\* p = new int(10); delete p; |

| **Compliant Code** |
| --- |
| This code uses a smart pointer to manage memory automatically. The smart pointer ensures that memory is properly released when it goes out of scope which prevents dangling pointers and improves memory safety. |
| std::unique\_ptr<int> p = std::make\_unique<int>(10); |

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

|  |
| --- |
| **Principles(s):**   * Keep It Simple (Principle 4)   Simple code is easier to understand and less vulnerable to memory errors.   * Practice Defense in Depth (Principle 8)   Memory safety is one layer of overall security.   * Adopt a Secure Coding Standard (Principle 10)   Following secure coding practices like using smart pointers and other memory safe practices prevents memory leaks. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.13 | memleak | This tool identifies unsafe memory operations that can lead to leaks. |
| Coverity | 2024.09 | USE\_AFTER\_FREE | This tool flags potential memory leaks. |
| Valgrind | 3.22 | Runtime detectors | This tool detects runtime memory leaks and invalid memory accesses. |
| AddressSanitizer | LLVM 17.0 | Runtime detectors | This tool provides dynamic checks for invalid memory access and leaks. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to catch programming errors during development, but disable them in production. |

| **Noncompliant Code** |
| --- |
| This code performs division without checking whether the denominator b is zero. Dividing by zero causes undefined behavior and could crash the program. |
| int divide(int a, int b) { return a / b; } |

| **Compliant Code** |
| --- |
| This code uses an assertion to ensure that the denominator b is not zero before performing division. This prevents divide by zero errors and helps maintain program stability. |
| #include <cassert> int divide(int a, int b) {  assert(b != 0);  return a / b; } |

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

|  |
| --- |
| **Principles(s):**   * Validate Input Data (Principle 1)   Catch logic and input errors through assertions.   * Use Effective Quality Assurance Techniques (Principle 9)   Supports proactive validation to catch errors early. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | misc-assert-side-effect | This tool warns about side effects in assertions that could affect program behavior. |
| cppcheck | 2.13 | unreachableCode | This tool detects assertions that cause unreachable code. |
| SonarQube | 10.5 | cpp:S6033 (Assert Misuse) | This tool flags assertions used incorrectly for production control flow. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Always handle exceptions properly to avoid system crashes and undefined states. |

| **Noncompliant Code** |
| --- |
| This code catches all exceptions but does nothing with them. Empty catch blocks silently ignore errors which can hide bugs, lead to undefined behavior or make debugging difficult. |
| try {  riskyOperation();  } catch (...) {} // empty catch block |

| **Compliant Code** |
| --- |
| This code catches exceptions of type std::exception and logs the error message. Handling exceptions appropriately ensures that errors are visible and can be addressed. This improves program reliability and maintainability. |
| try {  riskyOperation();  } catch (const std::exception& e) {  logError(e.what());  } |

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

|  |
| --- |
| **Principles(s):**   * Practice Defense in Depth (Principle 8)   Exception handling is part of layered security. Catching errors prevents them from compromising other layers.   * Adopt a Secure Coding Standard (Principle 10)   Using secure coding patterns for exception handling ensures resilience and prevents system failures. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | cpp:S2221 (Empty Catch Block) | This tool detects empty catch blocks that could hide errors. |
| Clang-Tidy | 17.0 | bugprone-empty-catch | This tool flags unhandled exceptions or improper handling patterns. |
| Coverity | 2024.09 | UNCAUGHT\_EXCEPT | This tool identifies missing error logging or unsafe exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Secure File Handling** | [STD-008-CPP] | Files should be opened with the least privilege required and file paths must be validated. |

| **Noncompliant Code** |
| --- |
| This code attempts to open a system file for writing. Modifying sensitive system files can compromise system integrity and security, and may require elevated privileges. |
| std::ofstream file("/etc/passwd"); |

| **Compliant Code** |
| --- |
| This code opens a local, non-sensitive file for writing, using the std::ios::trunc flag to safely overwrite existing content. This avoids the security risks associated with accessing system files. |
| std::ofstream file("user\_data.txt", std::ios::out | std::ios::trunc); |

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

|  |
| --- |
| **Principles(s):**   * Default Deny (Principle 5)   Unvalidated file paths should be denied by default.   * Adhere to the Principle of Least Privilege (Principle 6)   Restrict file access to the minimum required permissions. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.09 | PATH\_MANIPULATION | This tool detects unvalidated or unsafe file path usage. |
| Fortify SCA | 24.1 | Path Manipulation | This tool flags improper file access permissions. |
| SonarQube | 10.5 | cpp:S5545 (Unvalidated File Path) | This tool checks for missing file closures and validation to prevent data leakage. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Logging and Error Messages** | [STD-009-CPP] | Error messages should not expose sensitive information and logs must avoid leaking data. |

| **Noncompliant Code** |
| --- |
| This code prints the full SQL query when an error occurs which potential exposes sensitive information. Revealing query details can leak data or assist attackers in exploiting the system. |
| catch (SQLException e) {  std::cerr << e.getQuery() << std::endl;  } |

| **Compliant Code** |
| --- |
| This code provides a generic error message to the user while logging the detailed exception internally. This prevents sensitive data exposure while maintaining useful diagnostic information for developers. |
| catch (SQLException e) {  std::cerr << "Database error occurred" << std::endl;  logError(e.what());  } |

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

|  |
| --- |
| **Principles(s):**   * Architect and Design for Security Policies (Principle 3)   Logging should be designed in accordance to security policies to maintain secure operations.   * Sanitize Data Sent to Other Systems (Principle 7)   Sanitized logs and error messages prevent data leakage. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | cpp:S5542 (Information Exposure) | This tool detects logging of sensitive data such as credentials or queries. |
| Fortify SCA | 24.1 | Information Exposure | This tool flags risks of information exposure in logs. |
| Checkmarx | 9.6 | Log\_Information\_Exposure | This tool ensures logs and error messages do not leak confidential details. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Avoid Hardcoded Credentials** | [STD-010-CPP] | Never hardcode passwords, API keys or other sensitive credentials in code. |

| **Noncompliant Code** |
| --- |
| This code stores the API key directly in the source code. Hardcoding credentials is risky because anyone with access to the code can view and potentially misuse the credential. |
| const std::string API\_KEY = "12345SECRET"; |

| **Compliant Code** |
| --- |
| This code retrieves the API key from an environment variable rather than hardcoding it. Using secure storage mechanisms keeps credentials out of the source code, makes them easier to manage and reduces the risk of accidental exposure. |
| const std::string API\_KEY = std::getenv("API\_KEY"); |

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

|  |
| --- |
| **Principles(s):**   * Architect and Design for Security Policies (Principle 3)   Centralized secret management enforces security policies and supports safe operation.   * Adhere to Principle of Least Privilege (Principle 6)   Never grant access via embedded credentials in code. Retrieve secrets securely only when needed.   * Adopt a Secure Coding Standard (Principle 10)   Use secure coding patterns for handling credentials such as environment variables or secret management systems. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | cpp:S2068 (Hardcoded Credentials) | This tool scans source food for hardcoded strings that match common credential patterns. |
| Fortify Static Code Analyzer (SCA) | 24.1 | Hardcoded Password Rule | This tool identifies instances where credentials, encryption keys or authentication tokens are stored directly in source code. |
| GitGuardian CLI | 1.22 | Secrets Detection | This tool scans to detect secrets in repositories, commit history and CI pipelines. |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

In the Pre-production cycle, automation starts at the Assess and Plan and Design phases. At this stage, developers follow security driven practices such as applying OWASP standards and performing automated dependency scanning to identify vulnerabilities early in development. During the Build phase, automated tools including Cppcheck, Clang-Tidy and SonarQube analyze source code for issues such as data validation errors, unsafe memory operations and hardcoded credentials before code is finalized. In the Verify and Test phase, automated vulnerability scanning and unit testing validate both functionality and security compliance. Static and dynamic analysis tools ensure that all software meets Green Pace’s secure coding standards prior to deployment.

As the pipeline transitions into Production, automation supports the Transition and Health Check phase by performing secure configuration validation and automated penetration testing before release. Once deployed, continuous monitoring tools automatically track security events during the Monitor and Detect phase. Logging systems and analytics platforms automatically identify anomalies, potential threats and policy violations. Lastly, in the Respond and Maintain and Stabilize phases, automation enables rapid incident response and system recovery. Alerts trigger automated rollback procedures and continuous assessment ensures systems return to a secure and compliant baseline after an event. Through this integrated approach, Green Pace ensures that security automation is embedded throughout every stage of the software development lifecycle.

### 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** | Medium | Likely | Low | High | 4 |
| **STD-002-CPP** | High | Very Likely | Medium | Critical | 5 |
| **STD-003-CPP** | High | Very Likely | Medium | Critial | 5 |
| **STD-004-CPP** | Medium | Likely | Medium | High | 4 |
| **STD-005-CPP** | High | Very Likely | High | Critical | 5 |
| **STD-006-CPP** | Medium | Unlikely | Low | Medium | 3 |
| **STD-007-CPP** | High | Likely | Medium | High | 4 |
| **STD-008-CPP** | Medium | Likely | Medium | High | 4 |
| **STD-009-CPP** | Medium | Unlikely | Low | Medium | 3 |
| **STD-010-CPP** | High | Very Likely | High | Critical | 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 | All sensitive data stored in databases, backups or local file systems must be encrypted using industry standard algorithms such as AES-256. Encryption keys must be stored in a secure key management system and rotated periodically. This applies to all customer records, configuration files and log data that contain sensitive information. Database fields storing passwords or credentials must use salted and hashed encryption mechanisms. Encrypting data at rest prevents unauthorized access to stored information in the event of a data breach. |
| Encryption in flight | All data transmitted across internal and/or external networks must be encrypted using secure communication protocols such as HTTPS, SSH or VPN tunnels. Unencrypted transmission of credentials, API calls or sensitive data is prohibited. This policy applies whenever data is moving between systems. Developers must ensure that encryption certificates are current and configured properly to prevent man in the middle attacks. Encrypting data in transit protects information from interception and tampering. |
| Encryption in use | Sensitive data that is actively being processed in memory or by applications must be protected through secure runtime environments, container isolation and encrypted memory features when possible. Applications handling confidential data must minimize exposure in logs or cache. This policy applies during runtime operations such as data processing or analytics. Encrypting data in use ensures sensitive information stays secure even while being processed which reduces the risk of malware attacks or insider threats. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users must authenticate using secure credentials and multi-factor authentication (MFA). User identities must be verified before granting system access. This policy applies to user logins, access to development environments and database connections. Passwords must meet complexity standards and be changed regularly. API keys and tokens must be generated securely and rotated periodically. Strong authentication ensures only verified users and systems access Green Pace resources, protecting against unauthorized access. |
| Authorization | Access to data and system functions must follow the principle of least privilege. User permissions should be based on defined roles and access to sensitive systems must be explicitly granted and regularly reviewed. This policy applies to changes to the database, addition of new users and user-level access to files and resources. Proper authorization prevents abuse of privilege and ensures users access only the data necessary for their role. |
| Accounting | All user and system activities must be logged. Logs should capture authentication events, database changes, file access and privilege modifications. These logs should be stored securely for analysis. This policy applies across all Green Pace systems where users interact with applications or databases. Accounting ensures traceability in case of incidents. It provides evidence of compliance and supports rapid investigation of security events. |

**\***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 |  |
| **2.0** | 10/19/2025 | Completed Security Policy | Kenneth Wilkerson | Kenneth Wilkerson |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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