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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Ensuring all input data is validated protects against injection attacks, buffer overflows, and other vulnerabilities. Validation includes verifying data type, size, format, and acceptable value ranges to ensure input meets expected requirements. |
| 1. Heed Compiler Warnings | Compiler warnings frequently point to possible problems in the code that can result in functional errors or security flaws. TO preserve code quality and reduce risk, developers should take proactive measures to address these warnings. |
| 1. Architect and Design for Security Policies | From the start, software architecture and design should incorporate security. To reduce vulnerabilities, this involves utilizing secure frameworks, applying risk assessment methods, and ensuring security policies are followed. |
| 1. Keep It Simple | Complexity increases the risk of errors and vulnerabilities. Developers can design systems that are simpler to test, manage, and protect by following the simplicity principle. |
| 1. Default Deny | Unless specifically permitted, access should be denied by default in any system or application. This idea lowers the attack surface and minimizes unwanted access. |
| 1. Adhere to the Principle of Least Privilege | Only the permissions required to accomplish their tasks should be granted to users, applications, and processes. This reduces the possible harm from unintentional or malicious activity. |
| 1. Sanitize Data Sent to Other Systems | To prevent malicious inputs like SQL injection or cross-site scripting (XSS) from spreading to other areas of the system, all data sent to external systems should be sanitized. |
| 1. Practice Defense in Depth | By layering several security procedures, the system is protected even in the event of a failure. This includes network firewalls, encryption, and application-level security. |
| 1. Use Effective Quality Assurance Techniques | Software that is free of vulnerabilities and operates as intended is guaranteed by quality assurance. Peer reviews, penetration testing, and static code analysis are among the methods that are used to find and fix security vulnerabilities. |
| 1. Adopt a Secure Coding Standard | All developers will follow security best practices if a secure coding standard is established and followed. Consistency is promoted and vulnerabilities are decreased by standards like CERT C/C++ and MIRSA C. |

### 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-DAT] | This standard ensures the proper use of data types to prevent issues such as type mismatches, overflows, or precision errors. |

| **Noncompliant Code** |
| --- |
| Using inappropriate data types for variables can lead to precision loss or unexpected behavior. |
| Float total = 123456789.0; // May lose precision with large numbers  Int counter = -1; // Unsigned value incorrectly declared as signed |

| **Compliant Code** |
| --- |
| Select appropriate data types based on variable requirements |
| Double total = 123456789.0 // Maintains precision for large numbers  Unsigned int counter = 0; // Uses unsigned type for non-negative values. |

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

| **Principles(s):**  Validate Input Data - This principle emphasizes verifying data type, size, format, and acceptable value ranges to ensure the input meets expected requirements. Data Type Rationalization directly aligns with this principle because selecting appropriate data types ensures the data behaves as expected without causing precision loss, overflows, or unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | typeMismatch | Identifies mismatched types |
| Clang-tidy | 15.0 | performance | Flags inappropriate data use |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-VAL] | This standard ensures data values are checked before usage to avoid invalidation or out-of-range errors. |

| **Noncompliant Code** |
| --- |
| Failure to validate input values leads to unpredictable program behavior. |
| int divide(int a, int b) {  Return a /b; // Risk division by zero  } |

| **Compliant Code** |
| --- |
| Validate input values before performing operations |
| int divide(int a, int b) {  if (b == 0) {  throw std::invlaid\_argument(“Denominator cannot be zero”);  }  Return a / b;  } |

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

| **Principles(s):**  Validate Input Data - This principle emphasizes ensuring input values are validated for correctness, such as checking for acceptable ranges and avoiding invalid or harmful data usage. The coding standard about validating input values before performing operations aligns with this principle because it prevents unpredictable program behavior by ensuring all inputs meet expected requirements. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | divisionByZero | Detects division by zero issues |
| clang-tidy | 15.0 | bugprone | Identifies risky operations |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | This standard mitigates risks associated with unsafe string manipulation, such as buffer overflows. |

| **Noncompliant Code** |
| --- |
| Using unsafe functions for string operations can lead to vulnerabilities |
| char buffer[10];  strcpy(buffer, "This string is too long"); // Buffer overflow |

| **Compliant Code** |
| --- |
| Use safer alternatives for string operations |
| char buffer[10];  strncpy(buffer, "Safe", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Null-terminate |

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

| **Principles(s):**  Sanitize Data Sent to Other Systems - This principle emphasizes the importance of sanitizing data, including strings, to prevent malicious inputs from propagating. Proper string handling directly mitigates risks like buffer overflows, ensuring data integrity when interacting with external systems.  Adopt a Secure Coding Standard - Using safer alternatives for string operations aligns with adopting a secure coding standard, such as avoiding unsafe functions like strcpy and using secure functions like strncpy or standard library options. This reduces vulnerabilities and enforces best practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | befferOverflow | Detects buffer overflows |
| clang-tidy | 15.0 | security | Identifies unsafe string usage |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | this standard ensures input is sanitized to prevent SQL injection attacks |

| **Noncompliant Code** |
| --- |
| Directly concatenating user inputs into SQL queries is unsafe. |
| std::string query = "SELECT \* FROM users WHERE id = " + userInput; // Unsafe |

| **Compliant Code** |
| --- |
| Use parameterized queries to ensure input is sanitized. |
| std::string query = "SELECT \* FROM users WHERE id = ?";  sqlStatement.setInt(1, userId); // Uses parameterized query |

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

| **Principles(s):**  Validate Input Data – Ensuring all input is sanitized before being used in SQL queries directly relates to validating input data. By using parameterized queries, the input is constrained to expected formats and values, preventing injection attacks.  Sanitize Data Sent to Other Systems - This principle specifically addresses sanitizing data before sending it to external systems like databases. Parameterized queries enforce sanitization by ensuring that user inputs are treated as data rather than executable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| sqlmap | 1.6.11 | SQLInjection | Detects injection vulnerabilities |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | This standard addresses proper memory allocation, usage, and deallocation to prevent leaks and undefined behavior. |

| **Noncompliant Code** |
| --- |
| Failure to free allocated memory leads to memory leaks. |
| int\* ptr = new int[10]; // Memory not freed |

| **Compliant Code** |
| --- |
| Properly allocate memory after use. |
| int\* ptr = new int[10];  // Perform operations  delete[] ptr; // Frees memory |

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

| **Principles(s):**  Defense in Depth – Proper memory management ensures a layered defense by preventing common vulnerabilities like memory leaks and undefined behavior, which could weaken the system if exploited.  Adopt a Secure Coding Standard - Following secure coding standards (e.g., proper allocation and deallocation of memory) minimizes risks and promotes consistent practices for resource management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| valgrind | 3.19.0 | memoryLeaks | Detects memory management issues |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASS] | This standard ensures critical assumptions are explicitly checked in code, preventing unexpected behaviors during runtime. |

| **Noncompliant Code** |
| --- |
| Failing to assert critical assumptions can result in unexpected program behavior. |
| void process(int index, int size) {  int\* array = new int[size];  array[index] = 42; // May access out-of-bounds memory  } |

| **Compliant Code** |
| --- |
| Using assertions to ensure critical assumption are valid |
| #include <cassert>  void process(int index, int size) {  assert(index >= 0 && index < size); // Validate bounds  int\* array = new int[size];  array[index] = 42;  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques - Assertions are a fundamental quality assurance technique that enforces critical assumptions during runtime, ensuring program stability and reliability. This aligns with the principle of identifying and mitigating errors through effective testing and validation methods.  Keep it Simple - Using assertions simplifies error handling by explicitly defining critical assumptions, reducing the complexity of debugging and improving code maintainability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | assertationChecks | Flags missing assertions |
| clang-tidy | 15.0 | security | Identifies critical checks |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXP] | This standard ensures proper handling of exceptions to maintain program stability and security, preventing crashes or unidentified behavior. |

| **Noncompliant Code** |
| --- |
| Failing to catch and handle exceptions can result in unexpected program termination. |
| void processFile(const std::string& filename) {  std::ifstream file(filename);  file >> someData; // Crashes if file cannot be opened  } |

| **Compliant Code** |
| --- |
| Use exception handling to gracefully manage errors and recover when possible. |
| void processFile(const std::string& filename) {  try {  std::ifstream file(filename);  if (!file.is\_open()) {  throw std::ios\_base::failure("File cannot be opened");  }  file >> someData;  } catch (const std::ios\_base::failure& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  } |

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

| **Principles(s):**  Practice Defense in Depth – Ensures sensitive data is protected across multiple layers, reducing exposure risks.  Adopt a Secure Coding Standard – Encourages consistent use of industry standard encryption methods to protect ata integrity and confidentiality. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | exceptionChecks | Identifies missing exception handling |
| clang-tidy | 15.0 | errorHandling | Detects improper exception usage |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging and Auditing | [STD-008-LOG] | This standard ensures critical events are logged to enable auditing and facilitate troubleshooting. |

| **Noncompliant Code** |
| --- |
| Failing to log critical operations results in a lack of accountability and traceability. |
| void authenticateUser(const std::string& username, const std::string& password) {  if (username == "admin" && password == "password") {  // Authentication successful, no log  }  } |

| **Compliant Code** |
| --- |
| Log important events for security and debugging purposes. |
| void authenticateUser(const std::string& username, const std::string& password) {  if (username == "admin" && password == "password") {  std::cout << "User " << username << " authenticated successfully" << std::endl;  } else {  std::cerr << "Authentication failed for user " << username << std::endl;  }  } |

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

| **Principles(s):**  Practice Defense in Depth - Ensures sensitive data is protected across multiple layers, reducing the likelihood of exposure.  Adopt a Secure Coding Standard - Encourages consistent use of robust encryption methods to maintain data integrity and confidentiality. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | logingIssues | Detects missing logging |
| clang-tidy | 15.0 | securityLogging | Identifies log-related vulnerabilities |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety and Locking | [STD-009-LCK] | This standard ensures safe handling of shared resources in multi-threaded environments to prevent race conditions and deadlocks. |

| **Noncompliant Code** |
| --- |
| Failing to use proper synchronization can cause race conditions and inconsistent behavior. |
| void incrementCounter() {  counter++;  } |

| **Compliant Code** |
| --- |
| Use appropriate synchronization mechanisms to ensure thread safety. |
| #include <mutex>  std::mutex mtx;  void incrementCounter() {  std::lock\_guard<std::mutex> lock(mtx);  counter++;  } |

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

| **Principles(s):**  Adhere to the Principle of Least Privilege – Limits concurrent resource access to maintain consistency and security.  Adopt a Secure Coding Standard - Thread safety and locking mechanisms align with secure coding practices that emphasize consistency and prevent vulnerabilities in multi-threaded applications.  Practice Defense in Depth - Adding synchronization as a safeguard ensures robustness even in complex multi-threaded environments. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.11 | threadSafety | Detects synchronization issues |
| clang-tidy | 15.0 | concurrency | Identifies threading bugs |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Encryption Standards | [STD-010-ENC] | This standard ensures data is encrypted to maintain confidentiality and integrity during storage and transmission. |

| **Noncompliant Code** |
| --- |
| Storing sensitive data in plain text can lead to data breaches. |
| std::string password = "plaintext\_password"; |

| **Compliant Code** |
| --- |
| Use robust encryption techniques for sensitive data. |
| #include <openssl/evp.h>  std::string encryptPassword(const std::string& password) {  unsigned char key[EVP\_MAX\_KEY\_LENGTH] = "example\_key";  unsigned char iv[EVP\_MAX\_IV\_LENGTH] = "example\_iv";  unsigned char encrypted[128];  int encryptedLength;  EVP\_CIPHER\_CTX\* ctx = EVP\_CIPHER\_CTX\_new();  EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), nullptr, key, iv);  EVP\_EncryptUpdate(ctx, encrypted, &encryptedLength, (unsigned char\*)password.c\_str(), password.size());  EVP\_EncryptFinal\_ex(ctx, encrypted + encryptedLength, &encryptedLength);  EVP\_CIPHER\_CTX\_free(ctx);  return std::string((char\*)encrypted);  } |

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

| **Principles(s):**  Practice Defense in Depth – Ensures sensitive data is protected across multiple layers, reducing exposure risks.  Adopt a Secure Coding Standard – Encourages consistent use of industry-standard encryption method to protect data integrity and confidentiality. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OpenSSL | 3.0.0 | encryptionChecks | Validates secure encryption use |
| cppcheck | 2.11 | security | Detects insecure storage |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

**Integration Points in the DevSecOps Pipeline**

1. **Plan Phase**  
   Automation begins in the planning phase, where tools such as **threat modeling software** (e.g., OWASP Threat Dragon) are used to identify potential vulnerabilities and incorporate secure design principles. This ensures that security requirements are addressed early in the development cycle.
2. **Create Phase**  
   Developers will use IDE-integrated tools such as **SonarLint**, **Clang-Tidy**, or **Prettier** to enforce secure coding practices as they write code. These tools provide real-time feedback on potential issues, such as buffer overflows, type mismatches, or unsafe function usage, ensuring code quality from the outset.
3. **Verify Phase**  
   Automated static application security testing (SAST) tools like **Cppcheck** and **Clang Static Analyzer** will scan the codebase for compliance with the coding standards. These tools identify vulnerabilities such as memory leaks, missing input validation, or unsafe string handling before the application is deployed.
4. **Pre-Production Phase**  
   Dynamic application security testing (DAST) tools, including **OWASP ZAP** or **Burp Suite**, will test for runtime vulnerabilities in a controlled environment. Additionally, automated fuzzing tools can simulate unexpected inputs to ensure robust input validation and error handling.
5. **Release Phase**  
   Automated tools for build verification, such as **GPG (GNU Privacy Guard)**, will ensure that the software is properly signed and verified for authenticity and integrity before deployment. This step reduces the risk of deploying tampered or unauthorized builds.
6. **Prevent and Detect Phases**  
   At runtime, tools such as **Contrast Security** or **Runtime Application Self-Protection (RASP)** will monitor applications for suspicious activities. These tools dynamically detect and block threats such as SQL injection or buffer overflows in real time, ensuring ongoing protection.
7. **Respond and Adapt Phases**  
   Automated incident response tools, such as **Splunk Phantom** or **Cortex XSOAR**, will facilitate rapid responses to detected vulnerabilities. These tools automate log analysis, alert prioritization, and remediation actions, reducing response time. Lessons learned from incidents will feed into the continuous improvement of the pipeline.

### 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-001-DAT | High | Likely | Low | High | 5 |
| STD-002-VAL | High | Likely | Medium | High | 5 |
| STD-003-STR | High | Very Likely | Medium | High | 5 |
| STD-004-SQL | High | Very Likely | Low | High | 5 |
| STD-005-MEM | High | Likely | Medium | High | 5 |
| STD-006-ASS | High | Possible | Low | Medium | 4 |
| STD-007-EXP | High | Likely | Medium | High | 5 |
| STD-008-LOG | Medium | Likely | Low | Medium | 3 |
| STD-009-LCK | High | Very Likely | High | High | 5 |
| STD-010-ENC | High | Likely | High | 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 | **Policy:** All sensitive data stored within Green Pace's systems must be encrypted at rest using industry-standard encryption algorithms such as AES-256.  **Explanation:** Encryption at rest ensures that sensitive data, including files, databases, and backups, remains protected even if physical storage devices are compromised. This policy applies to all environments, including development, testing, and production, and is critical for meeting compliance standards like GDPR and HIPAA.  **Application:**   * Encrypt database files using transparent data encryption (TDE) tools. * Utilize file-level encryption for sensitive documents. * Ensure encryption keys are stored securely using a hardware security module (HSM). |
| Encryption in flight | **Policy:** All data transmitted between systems, users, and external applications must be encrypted using secure protocols such as TLS 1.2 or higher.  **Explanation:** Encryption in flight protects data during transmission, preventing interception or tampering by unauthorized parties. This policy applies to all internal and external communications, including API calls, emails, and web traffic.  **Application:**   * Enforce HTTPS for all web traffic. * Require secure email protocols (e.g., S/MIME or PGP). * Encrypt API communications using TLS. |
| Encryption in use | **Policy:** Sensitive data processed in memory must be encrypted or obfuscated to minimize risks of unauthorized access during runtime.  **Explanation:** Encryption in use ensures that sensitive data remains secure even when actively processed by applications. This policy applies to scenarios such as in-memory computations and cloud-based environments.  **Application:**   * Utilize secure enclaves or hardware-based trusted execution environments (TEEs). * Mask sensitive data in memory during processing. * Apply tokenization techniques for sensitive fields. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Policy:** All users and systems must authenticate using strong, multi-factor authentication (MFA) mechanisms.  **Explanation:** Authentication ensures that only authorized users and systems can access Green Pace's resources. This policy applies to user logins, API access, and administrative functions.  **Application:**   * Implement MFA for all user accounts. * Enforce secure password policies (e.g., minimum length, complexity requirements). * Use OAuth 2.0 or OpenID Connect for API authentication. |
| Authorization | **Policy:** Access to resources must be controlled using the principle of least privilege (PoLP) to ensure users and systems can access only what is necessary for their roles.  **Explanation:** Authorization governs what authenticated users or systems can do, minimizing the risk of unauthorized actions. This policy applies to role-based access control (RBAC) configurations, file access permissions, and database operations.  **Application:**   * Define and enforce role-based access controls (RBAC). * Regularly audit user permissions and revoke unnecessary access. * Use access control lists (ACLs) for file and system resources. |
| Accounting | **Policy:** All access and changes to resources must be logged and monitored for auditing purposes.  **Explanation:** Accounting ensures that all user activities are tracked, providing an audit trail for security investigations and compliance reporting. This policy applies to user logins, file access, and changes to sensitive configurations.  **Application:**   * Log all user login attempts and administrative actions. * Monitor file access and modifications with real-time alerts. * Use centralized logging tools like Splunk or ELK Stack for analysis. |

**\***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

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Principles Applied** | **Justification** |
| STD-001-DAT | 1, 9, 10 | Validating input data ensures correct data types are used. QA techniques and coding standards ensure reliability. |
| STD-002-VAL | 1, 5, 10 | Validating data values prevents undefined behavior and aligns with the "default deny" principle. |
| STD-003-STR | 7, 8, 10 | Sanitizing strings protects other systems. Defense in depth and secure coding practices reduce risks. |
| STD-004-SQL | 1, 7, 10 | Input sanitization prevents SQL injection, aligning with secure coding and sanitization principles. |
| STD-005-MEM | 8, 9, 10 | Memory protection adds layers of defense and ensures proper QA and adherence to secure coding standards. |
| STD-006-ASS | 4, 9, 10 | Assertions keep code simple, enforce QA techniques, and promote secure coding consistency. |
| STD-007-EXP | 3, 8, 10 | Exception handling incorporates secure design, defense in depth, and adherence to secure coding standards. |
| STD-008-LOG | 8, 9 | Logging and auditing provide multiple layers of defense and ensure quality assurance in troubleshooting. |
| STD-009-LCK | 6, 8, 10 | Adhering to least privilege ensures thread safety and adds defense in depth through secure coding practices. |
| STD-010-ENC | 8, 10 | Encryption applies defense in depth and ensures consistent use of secure coding standards. |

**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 | 12/15/2024 | Completed assignment | Hannah Hendrix |  |
| [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 |