**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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 21](#_Toc52464067)

[Coding Standard 10 23](#_Toc52464068)

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

[Project One 25](#_Toc52464070)

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

[2. Risk Assessment 25](#_Toc52464072)

[3. Automated Detection 25](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 26](#_Toc52464076)

[7. Map the Principles 28](#_Toc52464077)

[Audit Controls and Management 29](#_Toc52464078)

[Enforcement 29](#_Toc52464079)

[Exceptions Process 29](#_Toc52464080)

[Distribution 30](#_Toc52464081)

[Policy Change Control 30](#_Toc52464082)

[Policy Version History 30](#_Toc52464083)

[Appendix A Lookups 30](#_Toc52464084)

[Approved C/C++ Language Acronyms 30](#_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 | Unclean or unexpected input can introduce vulnerabilities. Always validate data to prevent buffer overflows, SQL injections, and logic errors. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential vulnerabilities. Resolve warnings to ensure code is free of common pitfalls like memory leaks and type mismatches. |
| 1. Architect and Design for Security Policies | Build security into the software architecture by identifying attack surfaces and minimizing exposure during the design phase. |
| 1. Keep It Simple | Simpler code is easier to debug, test, and secure. Avoid overly complex structures that may introduce hidden vulnerabilities. |
| 1. Default Deny | Deny all actions and permissions by default, only granting access or functionality as explicitly needed. |
| 1. Adhere to the Principle of Least Privilege | Grant the minimum level of access or permissions required for users or processes to perform their functions. |
| 1. Sanitize Data Sent to Other Systems | Ensure all outbound data is validated and sanitized to prevent injection attacks or data leaks. |
| 1. Practice Defense in Depth | Layer security mechanisms to create multiple barriers to entry, mitigating risks if one layer is compromised. |
| 1. Use Effective Quality Assurance Techniques | Employ rigorous testing methods, such as static analysis and fuzz testing, to identify and eliminate vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards (e.g., SEI CERT) to ensure consistent practices across teams and projects. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Proper Use of Data Types** | [STD-001-CPP] | Using incorrect or incompatible data types can lead to errors, undefined behavior, or security vulnerabilities. Use explicit types with clear boundaries to minimize these risks. |

| **Noncompliant Code** |
| --- |
| This code uses implicit type conversions, which can cause data loss or overflow. |
|  |

| **Compliant Code** |
| --- |
| This code ensures proper type alignment and prevents unintended conversions. |
|  |

| **Principles(s):**  **Validate Input Data:** Using incorrect or incompatible data types can lead to overflow or unexpected behavior. Ensuring explicit data types aligns with the principle of validating input data before processing.  **Adopt a Secure Coding Standard:** Using proper data types prevents undefined behavior, improving consistency and security.  **Heed Compiler Warnings:** Compiler warnings often highlight incorrect or unsafe type conversions that can lead to buffer overflows and data corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-dcl30-c | Detects improper use of data ranges |
| Coverity | Latest | OVERRUN | Identifies out-of-bounds errors |
| Cppcheck | Latest | arrayIndexOutOfBounds | Flags invalid array accesses |
| SonarQube | Latest | S2259 | Ensures proper range checking |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Validate Data Range** | [STD-002-CCP] | Validating that variables are within acceptable ranges prevents undefined behavior or logical errors. |

| **Noncompliant Code** |
| --- |
| The code does not validate user-provided data before use. |
|  |

| **Compliant Code** |
| --- |
| The code validates user input to ensure it is within bounds. |
|  |

| **Principles(s):**  **Validate Input Data:** Ensuring variables are within the correct range prevents overflows, underflows, and logic errors.  **Use Effective Quality Assurance Techniques:** Applying bounds checking helps prevent errors and ensures safe execution.  **Heed Compiler Warnings:** Warnings related to integer overflows or out-of-range values help enforce proper validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-dcl30-c | Detects improper use of data ranges |
| Coverity | Latest | OVERRUN | Identifies out-of-bounds errors |
| Cppcheck | Latest | arrayIndexOutOfBounds | Flags invalid array accesses |
| SonarQube | Latest | S2259 | Ensures proper range checking |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Strings are a common source of security vulnerabilities in software, especially in C++ where raw character arrays or improperly managed dynamic strings can lead to buffer overflows, null termination issues, or injection vulnerabilities. By using safe string-handling practices and libraries, developers can ensure strings are handled securely. |

| **Noncompliant Code** |
| --- |
| This code uses unsafe string handling with strcpy, which does not perform bounds checking, leading to potential buffer overflow. |
|  |

| **Compliant Code** |
| --- |
| This code uses strncpy, ensuring bounds checking and preventing buffer overflow by limiting the number of characters copied. |
|  |

| **Principles(s):**  **Sanitize Data Sent to Other Systems:** Properly handling strings prevents buffer overflows and injection attacks.  **Keep It Simple:** Using safe string functions reduces complexity and enhances security.  **Heed Compiler Warnings:** Compiler warnings flag unsafe string operations that may cause buffer overflows or null termination issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-str34-c | Detects unsafe string handling |
| Coverity | Latest | BUFFER\_OVERFLOW | Identifies buffer overflow risks |
| Cppcheck | Latest | strncat-check | Flags unsafe string concatenation |
| SonarQube | Latest | S2251 | Checks for insecure string handling |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection Prevention** | [STD-004-CPP] | SQL injection is a critical vulnerability that arises when user input is improperly handled in SQL queries. By using parameterized queries or prepared statements, developers can ensure that user input is treated as data and not as executable SQL, thus preventing injection attacks. |

| **Noncompliant Code** |
| --- |
| The following code dynamically constructs an SQL query using string concatenation, which is highly vulnerable to SQL injection attacks. |
|  |

| **Compliant Code** |
| --- |
| This code uses a generic parameterized query approach, where placeholders are used for user input, and the input is safely bound to these placeholders. |
|  |

| **Principles(s):**  **Sanitize Data Sent to Other Systems:** Ensures SQL queries are safe from user input manipulation.  **Default Deny:** Blocks SQL execution unless explicitly permitted through parameterized queries. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S3649 | Detects SQL injection vulnerabilities |
| SQLMap | Latest | N/A | Automated SQL injection testing |
| Coverity | Latest | SQL\_INJECTION | Identifies SQL query risks |
| Clang-Tidy | Latest | cert-env33-c | Ensures parameterized queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Proper memory management is critical in C++ to avoid issues like memory leaks, dangling pointers, or buffer overflows. This standard ensures that memory allocation, usage, and deallocation are performed safely. |

| **Noncompliant Code** |
| --- |
| Fails to release dynamically allocated memory, leading to memory leaks. |
|  |

| **Compliant Code** |
| --- |
| Uses std::unique\_ptr to ensure proper memory management. |
|  |

| **Principles(s):**  **Heed Compiler Warnings:** Many compilers provide warnings about uninitialized variables, memory leaks, and unsafe memory access, helping to catch security issues early.  **Adhere to the Principle of Least Privilege:** Proper memory management ensures that access to memory locations is controlled, reducing the risk of unauthorized memory modifications.  **Practice Defense in Depth:** Layering security controls such as memory protection mechanisms (e.g., stack canaries, ASLR, and DEP) ensures that even if one security layer fails, additional protections mitigate risks.  **Use Effective Quality Assurance Techniques:** Automated tools such as Valgrind, AddressSanitizer, and static analyzers can detect memory leaks, buffer overflows, and use-after-free vulnerabilities before deployment.  **Adopt a Secure Coding Standard:** Following best practices like using std::unique\_ptr and std::shared\_ptr in C++ prevents manual memory management errors and reduces security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | Latest | memcheck | Detects memory leaks |
| Clang-Tidy | Latest | cert-mem57-c | Flags improper memory allocation |
| Coverity | Latest | RESOURCE\_LEAK | Detects memory leaks |
| Cppcheck | Latest | memleak | Identifies memory leak risks |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CCP] | Assertions help ensure that the program’s state aligns with expected conditions during development. They should not be used for production error handling. |

| **Noncompliant Code** |
| --- |
| Uses an assertion for runtime error handling, which can be disabled in release builds. |
|  |

| **Compliant Code** |
| --- |
| Uses runtime checks instead of assertions for input validation. |
|  |

| **Principles(s):**  **Use Effective Quality Assurance Techniques:** Assertions help identify logic errors early.  **Architect and Design for Security Policies:** Ensures system stability through runtime validation.  **Heed Compiler Warnings:** Some compilers provide warnings about assertions that are always true or false, helping to catch logic errors early. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-err33-c | Checks improper use of assertions |
| Coverity | Latest | ASSERTION\_FAILURE | Identifies assertion misuses |
| Cppcheck | Latest | assert-usage | Ensures safe assertion usage |
| SonarQube | Latest | S3984 | Checks for assertions in production code |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions should be used to handle unexpected errors, not for regular control flow. This standard ensures exceptions are implemented in a consistent and maintainable way. |

| **Noncompliant Code** |
| --- |
| Catches a general exception without taking specific corrective actions. |
|  |

| **Compliant Code** |
| --- |
| Catches specific exceptions and provides meaningful handling. |
|  |

| **Principles(s):**  **Adopt a Secure Coding Standard:** Ensures error handling follows best practices.  **Keep It Simple:** Uses structured exception handling instead of inconsistent error codes.  **Heed Compiler Warnings:** Compiler warnings detect improper exception handling (e.g., catching general exceptions), improving program stability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-err58-cpp | Flags improper exception handling |
| Coverity | Latest | EXCEPTION\_HANDLING | Identifies exception misuse |
| Cppcheck | Latest | exception-safety | Ensures correct exception usage |
| SonarQube | Latest | S2737 | Checks for unsafe exception handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Logging | [STD-008-CPP] | Logs should never include sensitive information like passwords or personally identifiable information (PII) to avoid data breaches if logs are exposed. |

| **Noncompliant Code** |
| --- |
| Logs sensitive data, such as user passwords, in plain text. |
|  |

| **Compliant Code** |
| --- |
| Avoids logging sensitive data and masks critical details when necessary. |
|  |

| **Principles(s):**  **Default Deny:** Ensures sensitive data is never logged unless explicitly necessary.  **Practice Defense in Depth:** Encrypts logs and controls access to prevent unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S5334 | Identifies logging of sensitive data |
| Logwatch | Latest | N/A | Monitors log integrity |
| Coverity | Latest | SENSITIVE\_DATA | Flags sensitive logging issues |
| Clang-Tidy | Latest | cert-err33-c | Checks for unsafe logging practices |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | [STD-009-CPP] | User input must be validated and sanitized to ensure it meets expected formats and does not introduce vulnerabilities like buffer overflows or injections. |

| **Noncompliant Code** |
| --- |
| Fails to validate user input, allowing out-of-bounds access. |
|  |

| **Compliant Code** |
| --- |
| Validates the input length before processing. |
|  |

| **Principles(s):**  **Validate Input Data:** Ensuring that user input adheres to defined constraints prevents malicious exploitation such as buffer overflows and injection attacks.  **Sanitize Data Sent to Other Systems:** Cleaning and validating input prevents injection vulnerabilities that could compromise other components. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-str34-c | Checks for improper string handling and buffer overflows |
| Coverity | Latest | TAINTED\_SCALAR | Identifies unvalidated user input |
| Cppcheck | Latest | input-validation | Flags unsafe user input practices |
| SonarQube | Latest | S2631 | Detects improper input validation |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Code Comments and Documentation | [STD-010-CPP] | Clear comments and documentation ensure that code is maintainable, readable, and less prone to errors during future updates. |

| **Noncompliant Code** |
| --- |
| Code lacks meaningful comments, making it hard to understand. |
|  |

| **Compliant Code** |
| --- |
| Provides meaningful comments to explain the logic. |
|  |

| **Principles(s):**  **Adopt a Secure Coding Standard:** Clear documentation and well-structured comments help maintain secure coding practices across teams.  **Keep It Simple:** Well-documented code is easier to understand and less prone to errors when modified. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Doxygen | Latest | N/A | Generates documentation from code comments |
| Clang-Tidy | Latest | readability-comments | Ensures comments are useful and follow best practices |
| SonarQube | Latest | S1135 | Identifies missing or unclear comments |
| Cppcheck | Latest | style | Flags inconsistent documentation and formatting issues |

### 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



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.

**Pre-Production Automation**

1. **Assess and Plan:**
   * Use automated tools to scan for vulnerabilities and assess the regulatory landscape. Tools like **dependency-checkers** (e.g., OWASP Dependency-Check) can identify outdated or insecure dependencies.
   * Automate prioritization of security backlogs by integrating issue tracking tools (e.g., Jira with security plug-ins).
2. **Design:**
   * Automate security testing during design by employing **static application security testing (SAST)** tools like **SonarQube** or **Checkmarx** to identify flaws in early code designs.
   * Use templates to ensure adherence to OWASP and SEI CERT standards during design phases.
3. **Build:**
   * Implement automated **secure code scans** within CI/CD pipelines using tools like **GitHub Advanced Security** or **GitLab Security**.
   * Ensure **trusted repositories** for all open-source dependencies and enforce digital signatures to validate package integrity.
4. **Verify and Test:**
   * Integrate **vulnerability scanning** and **fuzz testing** into automated workflows. Tools like **Burp Suite**, **ZAP**, and **DAST solutions** can validate compliance and identify runtime vulnerabilities.
   * Use automated unit and integration testing frameworks (e.g., JUnit, PyTest) with security plugins to verify that code meets security benchmarks.

**Production Automation**

1. **Transition and Health Check:**
   * Automate penetration testing using tools like **Nessus** or **Metasploit** during the transition phase to validate security configurations.
   * Leverage configuration management tools like **Ansible** or **Chef** to ensure secure deployment practices.
2. **Monitor and Detect:**
   * Integrate **SIEM tools (e.g., Splunk, ELK Stack)** to continuously monitor logs, detect anomalies, and alert teams to potential threats.
   * Use intrusion detection systems (IDS) or runtime application self-protection (RASP) to detect and mitigate real-time threats.
3. **Respond:**
   * Implement automated response workflows using **security orchestration, automation, and response (SOAR)** platforms like **Palo Alto Cortex XSOAR** or **Splunk Phantom**.
   * Automate rollback mechanisms in case of a detected breach to maintain system integrity.
4. **Maintain and Stabilize:**
   * Use baseline management tools to regularly compare systems against defined security baselines and automatically remediate deviations.
   * Automate periodic security assessments and generate reports to maintain compliance with security policies.

### 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 | High | Medium | High | 2 |
| STD-003-CPP | High | High | Medium | High | 2 |
| STD-004-CPP | Critical | High | Medium | High | 1 |
| STD-005-CPP | High | High | Medium | High | 2 |
| STD-006-CPP | Medium | Medium | Low | Medium | 3 |
| STD-007-CPP | Medium | Medium | Low | Medium | 3 |
| STD-008-CPP | High | Medium | Low | High | 2 |
| STD-009-CPP | Critical | High | Medium | High | 1 |
| STD-010-CPP | Low | Low | Low | Medium | 4 |

### 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.

| **Encryption** | **Explanation and Best Practices** | **Policy Application** |
| --- | --- | --- |
| Encryption at Rest | Protects stored data by encrypting it on disk, databases, or storage media. Ensures that even if storage is compromised, the data remains inaccessible without the proper decryption key. Best Practices: AES-256 encryption, Key Management Service (KMS), proper access controls. | Policy: All sensitive data stored on company databases, servers, and backup storage must be encrypted using AES-256 or higher. Encryption keys must be securely managed using a dedicated KMS. Access to encrypted data must be logged. |
| Encryption in Flight (Data in Transit) | Protects data moving between systems, applications, or users over a network (e.g., HTTPS, TLS, VPNs). Prevents eavesdropping, man-in-the-middle (MITM) attacks, and data interception. Best Practices: TLS 1.3 or higher, SSH for remote access, IPsec for VPNs. | Policy: All data transmitted over public or private networks must use TLS 1.3, SSH, or IPsec. Internal API communication must be encrypted using mTLS. Traffic logs should be maintained to track data movement. |
| Encryption in Use | Protects actively processed data in memory or CPU registers. Prevents unauthorized access while data is being computed. Best Practices: Homomorphic encryption, secure enclaves (Intel SGX, AWS Nitro), RAM encryption. | Policy: Systems processing sensitive data must use secure enclaves or memory encryption technologies to prevent unauthorized access. Encryption keys must never be stored in plaintext in memory. Secure boot and runtime protections must be enforced. |

| **Triple-A Framework\*** | **Explanation and Best Practices** | **Policy Application** |
| --- | --- | --- |
| Authentication | Verifies user identity before granting access. Uses passwords, biometrics, MFA, and single sign-on (SSO). Best Practices: Use MFA, enforce password complexity, implement Zero Trust authentication. | Policy: All users must authenticate using multi-factor authentication (MFA) and single sign-on (SSO) where applicable. Default authentication methods should include biometrics, security keys, or hardware tokens where feasible. User login attempts must be logged and reviewed for anomalies. |
| Authorization | Determines what users can access once authenticated. Uses role-based access control (RBAC), least privilege, and access control lists (ACLs). Best Practices: Enforce least privilege, implement time-based access, review permissions regularly. | Policy: Access must be granted based on RBAC principles, enforcing the least privilege model. User access levels must be documented, and all new user additions must go through an approval process. |
| Accounting | Logs and monitors user activity for auditing and compliance. Uses SIEM tools, log monitoring, anomaly detection. Best Practices: Maintain centralized logging, monitor privileged users, enable alerting for unauthorized access. | Policy: All access logs must be collected and stored in a centralized logging system (e.g., Splunk, SIEM). All user logins, changes to databases, addition of new users, file access events, and privilege escalations must be logged. Privileged access logs must be reviewed regularly. Alerts must be triggered for unauthorized access attempts. |

**\***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** | **Linked Security Principles** | **Justification For Link** |
| Proper Use of Data Types (STD-001-CPP) | 1. Validate Input Data  10. Adopt a Secure Coding Standard  2. Heed Compiler Warnings | Ensuring proper data types prevents buffer overflows and logic errors. This aligns with validating input and following secure coding guidelines. |
| Validate Data Range (STD-002-CPP) | 1. Validate Input Data  9. Use Effective Quality Assurance Techniques  2. Heed Compiler Warnings | Validating input ensures values remain within acceptable limits, preventing undefined behavior. QA techniques like boundary testing help enforce this standard. |
| String Correctness (STD-003-CPP) | 7. Sanitize Data Sent to Other Systems  4. Keep It Simple  2. Heed Compiler Warnings | Unsafe string handling can lead to buffer overflows and injections. Sanitizing strings and using secure handling methods prevent security vulnerabilities. |
| SQL Injection Prevention (STD-004-CPP) | 5. Default Deny  7. Sanitize Data Sent to Other Systems | SQL injection is prevented by denying untrusted input and sanitizing data before sending it to the database. |
| Memory Protection (STD-005-CPP) | 2. Heed Compiler Warnings  6. Adhere to the Principle of Least Privilege  8. Practice Defense in Depth | Proper memory management restricts access to sensitive areas, preventing privilege escalation and unauthorized access to memory. |
| Assertions (STD-006-CPP) | 9. Use Effective Quality Assurance Techniques  3. Architect and Design for Security Policies  2. Heed Compiler Warnings | Assertions help catch errors early, preventing runtime failures. Designing with security policies ensures robust error handling mechanisms. |
| Exceptions (STD-007-CPP) | 10. Adopt a Secure Coding Standard  4. Keep It Simple  2. Heed Compiler Warnings | Using structured exception handling prevents unexpected crashes and improves maintainability. Following secure coding practices ensures consistency. |
| Secure Logging (STD-008-CPP) | 5. Default Deny  8. Practice Defense in Depth  3. Architect and Design for Security Policies | Sensitive data should not be logged unless explicitly needed. Encryption and access controls add an extra layer of defense. |
| Input Validation (STD-009-CPP) | 1. Validate Input Data  7. Sanitize Data Sent to Other Systems | Ensuring that input data meets expected formats prevents injection attacks and buffer overflows. Sanitization removes potential threats. |
| Code Comments and Documentation (STD-010-CPP) | 10. Adopt a Secure Coding Standard  4. Keep It Simple | Clear documentation improves code maintainability, ensuring secure practices are followed consistently. |

## 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 | 02/14/2025 | Submission of Initial Security Policy | Nicholas Kreuziger |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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