**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 | Any information coming into a system should be treated as untrusted until proven otherwise. Checking types, lengths, and formats makes sure only clean and expected data gets through, which prevents errors and attacks. |
| 1. Heed Compiler Warnings | Warnings exist for a reason, and ignoring them often means missing real problems. Paying attention to them helps catch bugs early and keeps unsafe code from slipping into production. |
| 1. Architect and Design for Security Policies | Security should be part of the design from day one, not something added at the end. Planning around trust boundaries and clear policies helps keep systems stronger against threats. |
| 1. Keep It Simple | The more complicated code becomes, the easier it is to hide mistakes. Keeping solutions straightforward makes them easier to understand, maintain, and secure. |
| 1. Default Deny | The more complicated code becomes, the easier it is to hide mistakes. Keeping solutions straightforward makes them easier to understand, maintain, and secure. |
| 1. Adhere to the Principle of Least Privilege | [Insert text.] Programs and users should only have the access they absolutely need. Limiting privileges keeps damage to a minimum if something goes wrong. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to files, logs, or databases, make sure it is cleaned and safe. This step prevents attackers from sneaking in harmful code or injecting commands. |
| 1. Practice Defense in Depth | No single security measure is enough. Layering protections—like input checks, permissions, and monitoring and creates backups if one control fails. |
| 1. Use Effective Quality Assurance Techniques | Testing, code reviews, and analysis tools are essential to find weaknesses. Using more than one method ensures problems are caught from different angles. |
| 1. Adopt a Secure Coding Standard | Following established secure coding guidelines keeps everyone on the same page. It also reduces the chances of introducing well known vulnerabilities into the codebase. |

### 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 | Use explicit and correct data types |

| **Noncompliant Code** |
| --- |
| Uses implicit narrowing that may wrap values. |
| int readValue();  std::array<int, 8> a{};  void f() {  unsigned char idx = readValue(); // implicit narrowing  a[idx] = 1; // out of range risk  } |

| **Compliant Code** |
| --- |
| Uses range checks and explicit conversions. |
| #include <array>  #include <optional>  std::optional<std::size\_t> toIndex(int v) {  if (v < 0 || v >= 8) return std::nullopt;  return static\_cast<std::size\_t>(v);  }  void f2(int v, std::array<int,8>& a) {  if (auto i = toIndex(v)) a[\*i] = 1;  } |

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

| Validate Input Data; Adopt a Secure Coding Standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | cpp:S1199 | Detects unsafe conversions and type narrowing |
| Cppcheck | 2.12 | typeCheck | Finds implicit conversions and undefined type behavior |
| Clang Static Analyzer | 17.0 | Core.CallAndMessage | Flags invalid casts or narrowing conversions |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Prevent arithmetic overflow and underflow |

| **Noncompliant Code** |
| --- |
| Performs math with no checks. |
| int add(int a, int b) { return a + b; } |

| **Compliant Code** |
| --- |
| Validates before adding. |
| #include <limits>  #include <optional>  std::optional<int> add\_checked(int a, int b) {  if (b > 0 && a > std::numeric\_limits<int>::max() - b) return std::nullopt;  if (b < 0 && a < std::numeric\_limits<int>::min() - b) return std::nullopt;  return a + b;  } |

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

| Validate Input Data; Use Effective QA Techniques. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | cpp:S3516 | Detects integer overflow and underflow operations |
| Cppcheck | 2.12 | INTEGER\_OVERFLOW | Identifies math operations exceeding data type limits |
| Clang Static Analyzer | 17.0 | overflowCheck | Detects potential arithmetic overflow conditions |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Use safe string handling |

| **Noncompliant Code** |
| --- |
| Writes beyond buffer limit. |
| char buf[8];  std::strcpy(buf, "too-long"); |

| **Compliant Code** |
| --- |
| Uses std: string for automatic safety. |
| #include <string>  std::string safeJoin(const std::string& a, const std::string& b) {  return a + b;  } |

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

| Validate Input Data; Defense in Depth. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | Buffer\_Overflow | Detects unsafe string functions and buffer overruns |
| Cppcheck | 2.12 | cpp:S1192 | Finds strcpy, strcat, and unsafe string manipulation |
| Clang Static Analyzer | 17.0 | bufferAccessOutOfBounds | Detects overflows caused by fixed sized char arrays |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | [Rationalize the standard.] |

| **Noncompliant Code** |
| --- |
| Builds query by string concatenation. |
| std::string user = getUserInput();  std::string sql = "SELECT \* FROM users WHERE name = '" + user + "';";  sqlite3\_exec(db, sql.c\_str(), nullptr, nullptr, nullptr); |

| **Compliant Code** |
| --- |
| Uses a parameterized query |
| sqlite3\_stmt\* stmt = nullptr;  sqlite3\_prepare\_v2(db, "SELECT \* FROM users WHERE name = ?1;", -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, user.c\_str(), -1, SQLITE\_TRANSIENT);  while (sqlite3\_step(stmt) == SQLITE\_ROW) { /\* read \*/ }  sqlite3\_finalize(stmt); |

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

| Sanitize Data Sent to Other Systems; Default Deny. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | cpp:S2077 | Detects concatenated SQL strings and injection risks |
| Fortify SCA | 23.2 | SQL\_Injection | Identifies unparameterized SQL queries |
| CodeQL | 2.16 | cpp/sql-injection | Scans for unsafe user input in SQL command construction |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Manage memory with RAII and smart pointers |

| **Noncompliant Code** |
| --- |
| Uses raw new without ownership |
| int\* make() { return new int(42); } |

| **Compliant Code** |
| --- |
| Uses std::unique\_ptr |
| #include <memory>  std::unique\_ptr<int> makeSafe() {  return std::make\_unique<int>(42);  } |

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

| Keep It Simple. Defense in Depth. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17.0 | DeadStores,MallocChecker | Detects leaks and invalied frees |
| Cppcheck | 2.12 | Memleak | Flags unfreed memory and missing ownership transfers |
| SonarQube | 10.4 LTS | Cpp:S3584 | Warns against manual new/delete usage |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Don’t use assertions for input validation |

| **Noncompliant Code** |
| --- |
| Uses assert on input |
| #include <cassert>  void setAge(int age) {  assert(age >= 0 && age <= 120);  } |

| **Compliant Code** |
| --- |
| Validates input at runtime. |
| #include <stdexcept>  void setAge(int age) {  if (age < 0 || age > 120) throw std::invalid\_argument("age out of range");  } |

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

| Validate Input Data. Use Effective QA Techniques |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | Cpp:S4124 | Detects misuse of asset statements for runtime input |
| Cppcheck | 2.12 | assertUsage | Flags asserts in input validation functions |
| Coverity Scan | 2024.1 | ASSERT\_SIDE\_EFFECT | Identifies non-deterministic asset conditions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle exceptions safely |

| **Noncompliant Code** |
| --- |
| Destructor throws. |
| struct F {  ~F() { throw std::runtime\_error("oops"); }  }; |

| **Compliant Code** |
| --- |
| Destructor cleans up without throwing. |
| struct F {  ~F() noexcept { /\* cleanup \*/ }  }; |

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

| Keep It Simple. Adopt a Secure Coding Standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | Cpp:S3626 | Detects destructors that throw exceptions |
| Clang Static Analyzer | 17.0 | C++ExceptionSpec | Flags noexept violations |
| Cppcheck | 2.12 | exceptionSafety | Ensures proper exception management in destructors |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Ownership and Lifetime | STD-008-CPP | Avoid dangling pointers |

| **Noncompliant Code** |
| --- |
| Returns reference to local. |
| const int& badRef() {  int x = 7;  return x;  } |

| **Compliant Code** |
| --- |
| Returns by value. |
| int goodVal() {  int x = 7;  return x;  } |

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

| Defense in Depth; Keep It Simple. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17.0 | DanglingReference | Detects references to freed or out of score memory |
| Cppcheck | 2.12 | Lifetime | Reports invalid lifetime of local refernces |
| Fortify SCA | 23.2 | Resource\_Lifetime | Check object lifetime consistency |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output Safety | STD-009-CPP | Use streams correctly with flush/seek and close files |

| **Noncompliant Code** |
| --- |
| Reads and writes with no repositioning. |
| #include <fstream>  void bad(std::fstream& f) {  int x; f >> x;  f << "next";  } |

| **Compliant Code** |
| --- |
| Clears and seeks before writing. |
| #include <fstream>  void good(const std::string& path) {  std::fstream f(path, std::ios::in | std::ios::out);  int x; f >> x;  f.clear(); f.seekp(0, std::ios::end);  f << "next";  } |

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

| Use Effective QA Techniques. Defense in Depth. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | Cpp:S3583 | Detects unclosed streams and incorrect stream positions |
| Cppcheck | 2.12 | fileHandling | Flags missing close() or steam flushes |
| Clang Static Analyzer | 17.0 | StreamChecker | Identifies unflushed or leaked file handles |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging and Format-String Safety | STD-010-CPP | Never use untrusted input as a format string |

| **Noncompliant Code** |
| --- |
| Uses raw input as format string. |
| void logUnsafe(const char\* userMsg) {  std::printf(userMsg);  } |

| **Compliant Code** |
| --- |
| Uses fixed format string. |
| #include <cstdio>  void logSafe(const char\* userMsg) {  std::printf("user: %s\n", userMsg);  } |

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

| Sanitize Data Sent to Other Systems; Defense in Depth. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 LTS | Cpp:S3458 | Detects format string vulnerabilities |
| Fortify SCA | 23.2 | Format\_String | Flags prinF-style misuse with user data |
| Cppcheck | 2.12 | formatString | Finds unsanitized format string arguments |

### 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.

Automation is a key part of keeping Green Pace’s development process secure. Tools like SonarQube, Fortify, and Cppcheck will run automatically as code is written and committed. These scans check for unsafe conversions, memory issues, and injection risks before the code can move forward. If problems are found, the build will stop until they are fixed.

Before release, automation continues with testing and code review to make sure all systems work safely together. After deployment, monitoring tools watch for unusual activity and alert the team if something looks wrong. This ongoing process keeps security active from start to finish and helps Green Pace stay compliant without slowing down development.

### 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 | Possible | Medium | High | 3 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | Critical | Possible | Medium | Very Likely | 5 |
| STD-005-CPP | High | Possible | Low | High | 3 |
| STD-006-CPP | Medium | Likely | Low | Medium | 3 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-008-CPP | High | Possible | Medium | High | 3 |
| STD-009-CPP | Medium | Likely | Low | Medium | 3 |
| STD-010-CPP | Critical | Likely | Medium | Very Likely | 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 | Protects stored data from being read if someone gains physical or unauthorized access to the system. Green Pace uses AES-256 encryption for databases, backups, and file storage. This keeps sensitive customer and company data safe even if a drive is stolen or breached. |
| Encryption in flight | Protects data while it moves between servers, users, and applications. Green Pace uses TLS 1.3 for all web and API traffic to prevent eavesdropping and man in the middle attacks. This ensures that all communication stays private and secure from start to finish. |
| Encryption in use | Protects data that is being processed in memory or inside virtual machines. This is done by isolating workloads and using secure containers. It helps prevent sensitive information from leaking while the system is running or being actively used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Confirms that a user or system is who they claim to be. Green Pace uses strong passwords, two factor authentication, and secure tokens to verify identity before allowing access. This protects against unauthorized entry into systems and applications. |
| Authorization | Controls what users are allowed to do after they log in. Access is based on role and job function so users only have the permissions they need. This prevents someone from viewing or changing data they are not responsible for. |
| Accounting | Keeps track of what users do once they are in the system. All actions are logged, including logins, data changes, and access to files. These logs help identify suspicious behavior and support audits if an incident occurs. |

**\***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

|  |  |  |
| --- | --- | --- |
| **Standard** | **Principles** | **Explanation** |
| **STD-001-CPP (Data Type)** | 1, 10 | Validating input and following secure coding standards help prevent unsafe type conversions and ensure code consistency. |
| **STD-002-CPP (Data Value)** | 1, 9 | |  | | --- | |  |  |  | | --- | | Input validation and strong quality assurance prevent overflow and underflow issues that can lead to crashes or corruption. | |
| STD-003-CPP (String Correctness) | 1, 8 | Proper validation and layered security controls keep strings from causing buffer overruns or injection flaws. |
| STD-004-CPP (SQL Injection) | 5, 7, 8 | Default deny policies, data sanitization, and layered defenses work together to stop SQL injection attacks. |
| STD-005-CPP (Memory Protection) | 4, 8 | Keeping code simple and using multiple layers of control strengthen memory safety and limit the damage of bugs. |
| STD-006-CPP (Assertions) | 1, 9 | Input validation and thorough testing catch invalid conditions without relying on risky assertions. |
| STD-007-CPP (Exceptions) | 4, 10 | Simple, secure coding standards help ensure that exceptions are handled safely without breaking program flow. |
| STD-008-CPP (Pointer Ownership and Lifetime) | 4, 8 | Using clear ownership rules and layered checks prevents dangling pointers and memory leaks. |
| STD-009-CPP (Input/Output Safety) | 8, 9 | Defense in depth and strong testing confirm that file and stream operations are handled securely. |
| STD-010-CPP (Logging and Format String Safety) | 7, 8 | Sanitizing data and layering defenses prevent attackers from exploiting format strings or unsafe log messages. |

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 10/12/2025 | Completed and updated coding standards, automation section, encryption policies, and Triple-A framework for Project One | Shawn Plaisted | Chief Information Security Officer (CISO) |
| 1.2 | 10/12/2025 | Final formatting, proofreading, and audit preparation | Shawn Plaisted | Chief Technology Officer (CTO) |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

Validate Input Data: Always treat external data as untrusted. Enforce strict type, length, range, and format checks at trust boundaries, and reject or sanitize anything that does not match expectations.

Heed Compiler Warnings: Turn on the highest warning levels and treat warnings as errors. Compilers and static analyzers often flag undefined behavior, dangerous casts, and lifetime issues that become vulnerabilities in production.

Architect and Design for Security Policies: Bake security into architecture: define trust boundaries, data flows, and policy enforcement points early. Make design choices that favor safe defaults, encapsulation, and clear ownership of resources and privileges.

Keep It Simple: Prefer simple, readable solutions over clever but fragile code. Simpler code has fewer hidden states and is easier to review, test, and harden over time.

Default Deny: Block by default and grant access only when explicitly allowed. This applies to network access, APIs, file operations, and user permissions.

Adhere to the Principle of Least Privilege: Run processes, services, and queries with the minimum rights they need. If a component is compromised, least privilege limits blast radius.

Sanitize Data Sent to Other Systems: Before emitting data to files, logs, shells, or databases, escape or parameterize it to prevent injection attacks or log forging.

Practice Defense in Depth: Use multiple layers of controls—validation, parameterization, output encoding, least privilege, auditing—so a single failure does not lead to compromise.

Use Effective Quality Assurance Techniques: Combine unit tests, fuzzing, static analysis, dynamic analysis, and code review. These practices catch defects that each technique alone may miss.

Adopt a Secure Coding Standard: Follow vetted guidance such as the SEI CERT C/C++ Coding Standard to avoid known dangerous constructs and enforce consistent, reviewable patterns.