**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 | To prevent injection, buffer overflows, and other behavior that may be unexpected, we need to ensure data types are checked thoroughly regarding input, along with their respective lengths and ranges. |
| 1. Heed Compiler Warnings | When a compiler gives you a warning, we need to treat them as potential errors to prevent unsafe practices. |
| 1. Architect and Design for Security Policies | From the start we need to consider security, including in our design. Implementing access restrictions, fault tolerances, and data flow control are paramount. |
| 1. Keep It Simple | Creates less spaghetti to comb through, as well as reducing the chance of bugs and security flaws. Keep the logic in your code as simple as you can to avoid potential holes. |
| 1. Default Deny | A system should by default deny access, only granting access to allowed operations. |
| 1. Adhere to the Principle of Least Privilege | This means every piece of your code or system should allow the least amount of privilege possible to reduce the impact of a breach. |
| 1. Sanitize Data Sent to Other Systems | Before any data gets sent out to an API, database, etc. sanitizing it beforehand prevents injections and potential information leaks. |
| 1. Practice Defense in Depth | The depth of your defense is like an onion, the more layers it has, the less likely it will be able to be completely penetrated by a malicious actor. |
| 1. Use Effective Quality Assurance Techniques | This will help catch bugs and vulnerabilities early by implementing code reviews, static analysis and other quality assurance tools. |
| 1. Adopt a Secure Coding Standard | Using standards such as SEI CERT C++ will promote safety, readability, and future maintainability of the process/program/system. This is a fool-proof way of making as robust a code as possible. |

### 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** | **Fixed-Width Integer Types Use** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Int and long vary in size across systems which will lead to overflow potential. By using fixed-width types like int32\_t, it ensures predictable behavior. |

| **Noncompliant Code** |
| --- |
| Uses int which may not have the same range on all systems that utilize the code. |
| int val = 1234567894561; // will cause overflow on 32-bit system |

| **Compliant Code** |
| --- |
| Uses int64\_t which specifies a 64-bit signed integer. |
| #include <cstdint>  int64\_t val = 1234567894561; |

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

| **Principles(s):** Validate Input Data – range checks prevent overflow/underflow.  Adopting Secure Coding Standards – aligns with SEI CERT INT rules for predictable widths. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17.0+ | cppcoreguidelines-narrowing-conversions, google-runtime-int | Flags narrowing/width-dependent integer use and non-portable int types. |
| Cppcheck | 2.17.1 | integerOverflow, wrongPrintfScanfArgNum | Detects potential integer overflow and suspicious integer use. |
| CodeQL (C/C++) | 2.x | Arithmetic overflow queries | Dataflow analysis to find expressions that can overflow given inputs. |
| UBSan (LLVM) | LLVM/Clang 17+ | -fsanitize=undefined | Runtime detection of signed/unsigned integer overflows in tests. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Numeric Input Range Validation** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Numeric values that are left unchecked can cause over/underflow. All external numeric data needs to be checked first. |

| **Noncompliant Code** |
| --- |
| No checking of user input before being used. For example, a division by zero or multiplication of a string. |
| int num = 12345 / userInput; |

| **Compliant Code** |
| --- |
| Checks the user input before being used. |
| if (userInput != 0) {  int num = 12345 / userInput;  } |

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

| **Principles(s):** Validate Input Data – checks bounds before use.  Default Deny – reject out-of-range/invalid inputs.  Adopt a Secure Coding Standard – formalizes numeric validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang static analyzer | 17.0+ | core.DivideZero, alpha.core.CallAndMessage | Finds division by zero and misuse of unchecked values. |
| clang-tidy | 17.0+ | bugprone-incorrect-roundings, bugprone-integer-division | Catches risky arithmetic dependent on unchecked inputs. |
| CodeQL (C/C++) | 2.x | Taint tracking for unvalidated input -> arithmetic/index use | Flags flow where external input reaches math/indices without checks. |
| SonarQube C++ | 10.x | Rules on validating input and avoiding undefined behavior | Raises issues for missing input validation and unsafe math. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness - Bounds** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | strcpy can cause buffer overflows due to being unbounded. Use alternative like strncpy\_s to prevent overflow. |

| **Noncompliant Code** |
| --- |
| Uses strcpy with no checking of bounds. |
| char age[25];  strcpy(age, userInput); |

| **Compliant Code** |
| --- |
| Uses strncpy\_s to limit string size. |
| char age[25];  strncpy\_s(age, sizeof(age), userInput, \_TRUNCATE); |

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

| **Principles(s):** Validate Input Data – verify lengths before copy.  Keep it Simple – prefer safer APIs.  Adopt a Secure Coding Standard – follow SEI STR rules. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17.0+ | cppcoreguidelines-pro-bounds-array-to-pointer-decay, hicpp-no-array-decay | Warns about array decays/bounds hazards with C-style strings. |
| Clang Static Analyzer | 17.0+ | security.insecureAPI.strcpy, security.insecureAPI.strcat | Flags unsafe C string APIs that enable overflow. |
| Cppcheck | 2.17.1 | bufferAccessOutOfBounds, nullPointer | Detects out-of-bounds copies and NUL-termination issues. |
| AddressSanitizer (ASan) | LLVM/Clang 17+ | -fsanitize=address | Runtime detection of buffer overflow/over-reads in tests. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Parameterized Queries** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prepared statements or parameterized queries can prevent untrusted input into SQL queries. |

| **Noncompliant Code** |
| --- |
| Adds input into SQL query string. |
| std::string userquerySQL = “SELECT user WHERE name = “ + userInput; |

| **Compliant Code** |
| --- |
| Parameterized query will separate code and data input. |
| sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, “SELECT user WHERE name = “, -1, &stmt, NULL);  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC); |

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

| **Principles(s):** Sanitize Data Sent to Other Systems – parameters separate code from data.  Default Deny – reject unsafe input and deny ad-hoc SQL.  Adhere to Least Privilege – database credentials limited to required operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL (C/C++) | 2.x | Taint tracking for string-built SQL | Flags flows where untrusted input is concatenated into SQL. |
| SonarQube C++ | 10.x | Injection vulnerability rules (RSPEC-3649 family) | Detects SQL built by concatenation; recommends prepared statements. |
| Semgrep | 1.x | c.cpp.security.injection.sql rules | Pattern-based detection of unsafe SQL construction. |
| gitleaks | 8.x | Built-in detectors (DB creds/DSNs) | Prevents checked-in credentials that amplify SQLi risk. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Use-After-Free** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Being able to access memory after it’s been deallocated allows for exploitation potential. Therefore, we need to nullify pointers after a delete. |

| **Noncompliant Code** |
| --- |
| Access memory after deletion. |
| int\* num = new int(54);  delete num;  std:: cout << \*num; |

| **Compliant Code** |
| --- |
| Sets the pointer to nullptr after deletion. |
| int\* num = new int(54);  delete num;  num = nullptr; |

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

| **Principles(s):** Adopt a Secure Coding Standard – RAII/nulling prevents UAF.  Keep it Simple – clear ownership/lifetime reduces errors.  Use Effective QA Techniques – sanitizers/tests catch violations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer (ASan) | LLVM/Clang 17+ | -fsanitize=address | Runtime detection of use-after-free and heap-buffer issues. |
| LeakSanitizer (LSan) | LLVM/Clang 17+ | -fsanitize=leak | Finds leaks that often co-occur with ownership bugs. |
| Cppcheck | 2.17.1 | memleak, useStlAlgorithm | Static patterns for leaks and safer STL usage. |
| Coverity | Latest | USE\_AFTER\_FREE, RESOURCE\_LEAK | Path-sensitive UAF/leak detection. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertion Side Effects** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertion side effects may not execute due to disabling in release build, resulting in inconsistent program state. |

| **Noncompliant Code** |
| --- |
| Side effects in assert. |
| assert(updateCount++); |

| **Compliant Code** |
| --- |
| Avoids side effects in the assert expression. |
| ++updateCount;  assert(updateCount > 0); |

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

| **Principles(s):** Heed Compiler Warnings – catch misuse of assert.  Adopt a Secure Coding Standard – asserts for invariants, not logic.  Use Effective QA techniques – validate behavior in release builds. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17.0+ | bugprone-assert-side-effect, misc-static-assert | Ensures assert() expressions have no side effects; prefer static\_assert when possible. |
| SonarQube C++ | 10.x | Exception handling correctness rules | Flags catching by value/slicing risks. |
| Compiler flags | Clang/GCC 13+ | -DNDEBUG build + CI diff | CI job verifies behavior parity between debug and release. |
| CodeQL (C/C++) | 2.x | Assertion with side effect query | Finds cases where assetions contain state-changing expressions that are skipped in release mode. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Catch by Reference** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catching an exception by reference is safer than by value due to the latter causing object slicing or other inefficiencies. |

| **Noncompliant Code** |
| --- |
| Exception catch by value. |
| try {  throw std::runtime\_error(“runtime”);  }  catch (std::exception e) {  std::cerr << e.what();  } |

| **Compliant Code** |
| --- |
| Exception catch by const reference. |
| try {  throw std::runtime\_error(“runtime”);  }  catch (const std::exception& e) {  std::cerr << e.what();  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – avoids slicing/undefined behavior.  Keep it Simple – consistent exceptions semantics. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17.0+ | cppcoreguidelines-avoid-non-const-global-variables, readability-const-return-type | Promotes const-correctness and immutability for static storage. |
| SonarQube C++ | 10.x | Rules encouraging const for globals/statics | Flags mutable global state that can be made const. |
| Cppcheck | 2.17.1 | constVariable, unusedFunction | Suggests const where possible, reducing accidental writes. |
| GCC/Clang compiler warnings | GCC 11+ / Clang 13+ | -Wcatch-value | Warns when exceptions are caught by value; enforces throw by value, catch by const reference. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Declare Static Objects as const** |
| --- | --- | --- |
| Declarations and Initialization | [STD-008-CPP] | Static storage may be modified if not for const, which could potentially introduce security issues and other effects on the program. (SEI CERT DCL30-C) |

| **Noncompliant Code** |
| --- |
| Unintended modification due to declaration of static pointer without const. |
| static char\* message = “Hello World!”; |

| **Compliant Code** |
| --- |
| Declare pointer as const to prevent accidental changes. |
| static const char\* message = “Hello World!”; |

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

| **Principles(s):** Default Deny – immutable by default to prevent accidental writes.  Architect and Design for Security Policies – design for immutability where possible.  Adopt a Secure Coding Standard – SEI DCL guidance on const-correctness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17.0+ | cppcoreguidelines-avoid-non-const-global-variables, readability-const-return-type | Promotes const-correctness and immutability for static storage. |
| SonarQube C++ | 10.x | Rules encouraging const for globals/statics | Flags mutable global state that can be made const. |
| Cppcheck | 2.17.1 | constVariable, unusedFunction | Suggests const where possible, reducing accidental writes. |
| Clang compiler diagnostics | 13+ | -Wglobal-constructors, -Wexit-time-destructors | Flags non-trivial global/static initialization and teardown; encourages making eligible statics const/constexpr or tightening scope to avoid mutable globals. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Avoid Unknown Function Calling** |
| --- | --- | --- |
| Concurrency | [STD-009-CPP] | Calling untrusted code while holding a lock can lead to deadlocks or priority inversion, weakening system reliability, and performance. (SEI CERT CON37-C) |

| **Noncompliant Code** |
| --- |
| Calls function pointer that may block or acquire its own locks. |
| std::mutex m;  void (\*callback)();  void process() {  std::lock\_guard<std::mutex> lock(m);  callback();  } |

| **Compliant Code** |
| --- |
| Releases the lock before calling the potentially unsafe function. |
| std::mutex m;  void (\*callback)();  void process() {  {  std::lock\_guard<std::mutex> lock(m);  }  callback();  } |

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

| **Principles(s):** Architect and Design for Security Policies – structured locking/hand-off design.  Practice Defense in Depth – reduce deadlock/priority inversion risk.  Use Effective QA Techniques – analysis/tests verify lock discipline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Thread Safety Analysis | 17.0+ | -Wthread-safety | Statically verifies lock/unlock disciple and call patterns. |
| ThreadSanitizer (TSan) | LLVM/Clang 17+ | -fsanitize=thread | Runtime detection of data races; can reveal lock misuse patterns. |
| Cppcheck | 2.17.1 | deadlock, knownConditionTrueFalse | Heuristics for deadlock risks and suspicious lock usage. |
| Coverity | Latest | Concurrency/locking checkers | Path-sensitive analysis of lock held across external/unknown calls. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Exclude User Input from Format Strings** |
| --- | --- | --- |
| Input/Output | [STD-010-CPP] | printf or fprintf can be exploited due to the nature of user-controlled format strings in the aforementioned functions. This can cause the read/write of arbitrary memory, therefore we need to separate data from the format string. (SEI Cert FIO30-C) |

| **Noncompliant Code** |
| --- |
| Passes user input directly as a format string. |
| printf(userInput); |

| **Compliant Code** |
| --- |
| Uses a fixed format string and passes user input as a separate argument. |
| printf(“%s”, userInput); |

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

| **Principles(s):** Sanitize Data Sent to Other Systems – never treat input as a format string.  Validate Input Data – check/normalize external input before printing.  Adopt a Secure Coding Standard – follow SEI FIO rules on format safety. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compiler warnings (Clang/GCC) | 13+ | -Wall -Wextra -Wformat -Wformat-security | Detects format string mismatches and user-controlled format risks. |
| clang-tidy | 17.0+ | bugprone-format, hicpp-vararg | Warns about unsafe varargs/format usage patterns. |
| CodeQL (C/C++) | 2.x | Uncontrolled format string queries | Flags flows where input reaches the format parameter. |
| SonarQube C++ | 10.x | Format-string safety rules | Enforces fixed format strings with data as separate args. |

### 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.

The company Green Pace enforces these above standards with automated controls integrated across the DevSecOps pipeline, where tools run early and continuously, with quality enforcement that block releases on high-risk findings. The DevSecOps diagram highlights where each class of standard is best enforced. During early planning and threat modeling, security requirements for each coding standard are translated into automated checks. An example would be the need for fixed-width integer types (STD-001) and validated numeric ranges (STD-002) is documented as linting and static analysis rules that will be applied in later stages. Security-focused design reviews include automated scanning of architecture documentation, ensuring database access patterns use parameterized queries (STD-004) and concurrency designs avoid unsafe locking (STD-009).

At build time, automation enforces compiler flags, runs clang-tidy, Cppcheck, and CodeQL with profiles mapped to the 10 standards, and blocks merges if critical issues are found. Standards like memory safety (STD-005) and safe exception handling (STD-007) are enforced here. Functional and security tests run with sanitizers and fuzzers to detect runtime violations of standards, especially buffer safety (STD-003), format string safety (STD-010), and use-after-free issues (STD-005). Automated SQL injection tests verify compliance with STD-004. Before deployment, penetration tests and automated configuration scans verify that no code violating the standards remains. Database queries are reviewed automatically to ensure parameterization, and any mutable global/static objects (STD-008) are flagged for review.

Runtime monitoring tools, log analysis, and intrusion detection are configured to alert on anomalies that could indicate a coding standard breach in production like unusual exception patterns (STD-007). Automated re-scanning of the codebase on release branches ensures ongoing compliance. If a breach is detected, automation supports rapid rollback to a secure baseline. If automation detects violations or security incidents linked to these standards, the response process includes blocking further deployments, rolling back affected services, and triggering code audits. With this plan, Green Pace ensures that security compliance is a continuously automated safeguard throughout the software lifecycle.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | Very High | 1 |
| STD-003-CPP | High | Possible | Medium | High | 2 |
| STD-004-CPP | Critical | Possible | Medium | Very High | 1 |
| STD-005-CPP | High | Possible | Medium | High | 2 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | Medium | Possible | Low | Medium | 3 |
| STD-008-CPP | Medium | Possible | Low | Medium | 3 |
| STD-009-CPP | High | Possible | Medium | High | 2 |
| STD-010-CPP | High | Likely | Medium | Very High | 1 |

### 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 | Protecting stored data from unauthorized access by encrypting it on a disk, in databases, and in backups. All sensitive data, including customer information, intellectual property, and authentication credentials, must be encrypted at rest using AES-256 or stronger. Disk level encryption and database transparent data encryption are required. This ensures confidentiality even if storage media is lost, stolen, or accessed without authorization. It is applied whenever data is stored on servers, on developer workstations, portable media, or backup systems. |
| Encryption in flight | This is protecting data as it travels across networks from interception or tampering. All communication carrying sensitive or control data must use TLS 1.3 with strong cipher suites. Mutual TLS is required for service-to-service traffic where possible. Certificates must be managed via a trusted PKI and rotated regularly. This prevents man-in-the-middle attacks and data leakage. It applies for all external and internal communications over potentially insecure networks. |
| Encryption in use | Protecting sensitive data while it is being processed in memory. Applications should minimize the time sensitive data remains in plaintext form in memory. Use operating system-level protections and zeroize buffers after use. This limits exposure if memory is read by malicious processes or through vulnerabilities. This applies whenever processing sensitive data such as passwords, API keys, tokens, or personally identifiable information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies the identity of a user or system before granting access. All access to Green Pace systems must require strong authentication. This includes multi-factor authentication for administrative accounts and short-lived tokens for services. No hard-coded credentials are permitted. All authentication logs are maintained for audit purposes. It applies whenever a user, service, or device attempts to access a protected system, network, or data resource. |
| Authorization | Grants an authenticated entity permission to perform specific actions or access specific resources. Authorization must follow the principle of least privilege. Roles and permissions are assigned based on job function, and access is denied by default unless explicitly granted. Temporary elevation of privileges must be time-bound and logged, which minimizes potential damage from compromised accounts. It applies after authentication, before granting access to any protected function, dataset, or system. |
| Accounting | Tracks and records user and system activity to provide an audit trail. All critical operations like logins, database changes, new user creation, and file access must be logged with timestamp, user ID, source, and outcome. Logs must be forwarded to a centralized, tamper-evident logging system for correlation and long-term retention. It applies continuously in all environments, with heightened monitoring for production systems and administrative actions. |

**\***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.

|  |  |  |
| --- | --- | --- |
| **Standard** | **Principle Numbers** | **Justification** |
| STD-001-CPP – Fixed-Width Integer Types Use | 1, 10 | 1. Validate Input Data – prevents overflow/underflow from unexpected values.   (10) Adopt a Secure Coding Standard – aligns with SEI CERT INT rules for predictable widths. |
| STD-002-CPP – Numeric Input Range Validation | 1, 5, 10 | (1) Validate Input Data – enforces range checks.  (5) Default Deny – reject out-of-range values.  (10) Adopt a Secure Coding Standard – formalizes numeric validation practices. |
| STD-003-CPP – String Correctness – Bounds | 1, 4, 10 | (1) Validate Input Data – ensures lengths are safe.  (4) Keep It Simple – safer API use reduces complexity.  (10) Adopt a Secure Coding Standard – aligns with SEI STR rules. |
| STD-004-CPP – Parameterized Queries | 7, 5, 6 | (7) Sanitize Data Sent to Other Systems – parameters separate code from data.  (5) Default Deny – reject unsafe input.  (6) Adhere to Least Privilege – limit DB permissions. |
| STD-005-CPP – Use-After-Free | 10, 4, 9 | (10) Adopt a Secure Coding Standard – RAII/nulling avoids UAF.  (4) Keep It Simple – clear lifetime rules.  (9) Use Effective QA Techniques – sanitizers/tests detect violations. |
| STD-006-CPP – Assertion Side Effects | 2, 10, 9 | (2) Heed Compiler Warnings – avoid side-effects in asserts.  (10) Adopt a Secure Coding Standard – asserts for invariants, not logic.  (9) Use Effective QA Techniques – verify in both debug/release. |
| STD-007-CPP – Catch by Reference | 10, 4 | (10) Adopt a Secure Coding Standard – avoids slicing/UB.  (4) Keep It Simple – standardizes exception handling. |
| STD-008-CPP – Declare Static Objects as const | 5, 3, 10 | (5) Default Deny – immutability prevents changes by default.  (3) Architect and Design for Security Policies – design with const-correctness.  (10) Adopt a Secure Coding Standard – aligns with SEI DCL guidance. |
| STD-009-CPP – Avoid Unknown Function Calling | 3, 8, 9 | (3) Architect and Design for Security Policies – structure lock discipline.  (8) Practice Defense in Depth – prevent cascading concurrency issues.  (9) Use Effective QA Techniques – analysis/tests verify locking. |
| STD-010-CPP – Exclude User Input from Format Strings | 7, 1, 10 | (7) Sanitize Data Sent to Other Systems – never treat input as a format string  (1) Validate Input Data – check/normalize input before printing.  (10) Adopt a Secure Coding Standard – follows SEI FIO rules. |

**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 | 08/09/2025 | Completed Security Policy | Kris Marchevka | Professor Singh |
| [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 |