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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_b6xjsf7si6b)

[Purpose 2](#_jylrpj873sy0)

[Scope 2](#_matfiwf04h97)

[Module Three Milestone 2](#_bw494r4bm4r6)

[Ten Core Security Principles 2](#_wy7973xt9wpw)

[C/C++ Ten Coding Standards 3](#_gtzuxq30r1z1)

[Coding Standard 1 4](#_ex8wdc9mmwud)

[Coding Standard 2 5](#_koccg995kzf)

[Coding Standard 3 6](#_f2frdcjk4o2v)

[Coding Standard 4 7](#_1x0ipn1vvvsa)

[Coding Standard 5 8](#_y9her64m8ift)

[Coding Standard 6 9](#_audmpq8y0m4)

[Coding Standard 7 10](#_w39jyvyztz89)

[Coding Standard 8 11](#_ttxunsw6skng)

[Coding Standard 9 13](#_t8cuwtuy7w2h)

[Coding Standard 10 14](#_81w5wybw0rwj)

[Defense-in-Depth Illustration 15](#_m4y4a4peg96k)

[Project One 15](#_eie300o6s52n)

[1.](#_ivqq615i75cn) Revise the C/C++ Standards 15

[2.](#_zdmfdovdkshi) Risk Assessment 15

[3.](#_7igktlxi02aj) Automated Detection 15

[4.](#_6er4zblwylqr) Automation 15

[5.](#_396akxdtocf3) Summary of Risk Assessments 16

[6.](#_1b9gpowqsb9m) Create Policies for Encryption and Triple A 16

[7.](#_4iuloefmdzq1) Map the Principles 17

[Audit Controls and Management 18](#_a695u9om8rk4)

[Enforcement 18](#_xvm6x8ukrtus)

[Exceptions Process 18](#_12dy83dl6hg7)

[Distribution 19](#_s6b13sij5em)

[Policy Change Control 19](#_klf5z26cc1vr)

[Policy Version History 19](#_j3ika7ajd2cv)

[Appendix A Lookups 19](#_9nuuy0g06kxc)

[Approved C/C++ Language Acronyms 19](#_56z7yzxxffpd)

## 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 | Always assume external input is malicious until proven otherwise. Every interface, command line, GUI field, config file, socket, or API, must check length, type, range, and format before the data is used. Reject or sanitize anything that violates the contract to prevent injection, crashes, or logic errors. |
| 1. Heed Compiler Warnings | Enable the strictest warnings (e.g., -Wall -Wextra -Wpedantic) and treat them as build-breaking errors. Modern compilers can spot type truncation, uninitialized variables, and suspicious casts long before code reaches production, eliminating whole classes of defects at zero runtime cost. |
| 1. Architect and Design for Security Policies | Security cannot be bolted on. Choose designs (layered architecture, privilege separation, defense-in-depth) that naturally enforce the organization’s policy goals: confidentiality, integrity, availability, and auditability. |
| 1. Keep It Simple | Complexity hides bugs and widens the attack surface. Favor small functions, clear data flows, and well-known algorithms over clever tricks. Simple code is easier to test, reason about, and secure. |
| 1. Default Deny | When a decision is ambiguous, fail closed. If a file path, port, or permission is not explicitly allowed, block it. This limits damage when new resources appear or validation logic fails. |
| 1. Adhere to the Principle of Least Privilege | Give each process and thread only the permissions it needs for its job, and no more. Dropping privileges (e.g., setuid(0) → non-root) reduces the impact of a compromise. |
| 1. Sanitize Data Sent to Other Systems | Outgoing data can become someone else’s incoming attack. Escape special characters in SQL, HTML, JSON, and shell commands to prevent cross-system exploits and maintain data integrity. |
| 1. Practice Defense in Depth | Combine multiple overlapping controls: input validation, ASLR, stack canaries, sandboxing, and runtime monitoring. If one layer fails, another arrests the attack. |
| 1. Use Effective Quality Assurance Techniques | Pair manual code reviews with static analysis, fuzzing, unit tests, and continuous integration. Each technique finds different bug classes, and together they keep regressions out of production. |
| 1. Adopt a Secure Coding Standard | Follow a vetted rule set (e.g., SEI CERT C++) to unify terminology, clarify expectations, and provide ready-made examples of compliant vs. non-compliant code for your team. |

### 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 | Name: Avoid signed/unsigned mix when comparing or assigning (INT30-C).  Rationale: Mixing signed and unsigned values can wrap or mis-compare, producing logic errors and exploitable overflows. |

| **Noncompliant Code** |
| --- |
| Compares a negative int with size\_t; the int is converted to a huge unsigned value. |
| int read\_len = getLength(); // may return –1 on failure  size\_t buf\_sz = 256;  if (read\_len < buf\_sz) { // UB: read\_len promoted to size\_t  char buf[256];  readData(buf, read\_len);  } |

| **Compliant Code** |
| --- |
| Uses the same signed type for both operands, then casts only after validation. |
| int read\_len = getLength();  const int BUF\_SZ = 256;  if (read\_len >= 0 && read\_len < BUF\_SZ) {  char buf[BUF\_SZ];  readData(buf, static\_cast<size\_t>(read\_len));  } |

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

| **Principles(s):** 2 – Heed Compiler Warnings & 10 – Adopt a Secure Coding Standard: strict warnings catch mixed-sign maths early, and INT30-C formalises the rule. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 18.1.1 | bugprone-signed-unsigned-comparison | warns on |
| SonarQube | 2025.1 LTA | cpp:S845 | flags implicit sign-change in expressions |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Name: Check array bounds before access (ARR38-C)  Rationale: Prevents out-of-bounds reads/writes that crash or corrupt memory. |

| **Noncompliant Code** |
| --- |
| Index from user is used directly without range checking. |
| int idx = std::stoi(argv[1]);  int val = arr[idx]; // may access out of range |

| **Compliant Code** |
| --- |
| Validates index against the container’s size before access. |
| int idx = std::stoi(argv[1]);  if (idx >= 0 && idx < static\_cast<int>(arr.size()))  val = arr[idx];  else  std::cerr << "Index out of range\n"; |

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

| **Principles(s):** 1 – Validate Input Data & 3 – Architect for Security: bounds checks enforce safe design and stop attacker-chosen indices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 18.1.1 | clang-analyzer-core.ArrayOutOfBounds | Detects possible out-of-range access. |
| SonarQube | 2025.1 | cpp:S3518 (array index out of bounds) | Highlights unguarded subscripts. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Name: Replace unsafe string functions (STR03-C)  Rationale: strcpy, sprintf, etc. ignore buffer length and enable classic overflows. |

| **Noncompliant Code** |
| --- |
| Copies arbitrary-length input into a fixed 16-byte buffer. |
| char dst[16];  strcpy(dst, src); // overflow if src ≥ 16 |

| **Compliant Code** |
| --- |
| Uses strncpy with explicit bound and terminates buffer. |
| char dst[16];  strncpy(dst, src, sizeof(dst) - 1);  dst[sizeof(dst) - 1] = '\0'; |

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

| **Principles(s):** 1 – Validate Input Data & 7 – Sanitize Data Sent to Other Systems: safe string APIs block overflow and keep outbound data clean. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 18.1.1 | security.insecureAPI.strcpy | Flags uses of unsafe string functions |
| SonarQube | 2025.1 | cpp:S1036 | Detects unchecked strcpy/sprintf calls. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Name: Use parameterized queries (FIO30-C)  Rationale: Separates code from data, eliminating injection vectors. |

| **Noncompliant Code** |
| --- |
| Builds SQL by concatenating unsanitized user input. |
| std::string sql =  "SELECT \* FROM users WHERE name='" + userInput + "'";  db.exec(sql); |

| **Compliant Code** |
| --- |
| Pre-compiles statement and binds user input as a parameter. |
| sqlite3\_stmt\* stmt = nullptr;  sqlite3\_prepare\_v2(db,  "SELECT \* FROM users WHERE name=?", -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1,  userInput.c\_str(), -1, SQLITE\_TRANSIENT);  sqlite3\_step(stmt); |

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

| **Principles(s):** 7 – Sanitize Data Sent to Other Systems & 5 – Default Deny: parameterised queries isolate user data; anything not bound is rejected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 2025.1 | cpp:S3649 | Spots string-built SQL vulnerable to injection |
| OWASP ZAP | 2.15.0 | “SQL Injection” active scan rule | Runtime DAST proof of exploitability. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Name: Free dynamically allocated memory exactly once (MEM31-C)  Rationale: Double-free corrupts the heap; leaks exhaust resources. |

| **Noncompliant Code** |
| --- |
| Deletes the same pointer twice. |
| int\* p = new int[10];  delete[] p;  delete[] p; // double free |

| **Compliant Code** |
| --- |
| Transfers ownership to RAII smart pointer; memory freed once automatically. |
| std::unique\_ptr<int[]> p(new int[10]); |

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

| **Principles(s):** 8 – Defense in Depth & 9 – Effective QA: RAII removes double-free at design time; runtime tools catch leaks that slip through. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.23 | memcheck | Finds leaks, use-after-free, double-free. |
| clang-tidy | 18.1.1 | clang-analyzer-cplusplus.NewDeleteLeaks | Static double-delete detection. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Name: Do not rely on assert() for essential checks (EXP12-C)  Rationale: Assertions disappear when NDEBUG is defined; critical tests must execute in release builds. |

| **Noncompliant Code** |
| --- |
| Dereferences a pointer relying solely on assert to ensure non-null. |
| assert(ptr != nullptr);  \*ptr = 42; // may crash in release |

| **Compliant Code** |
| --- |
| Performs a runtime check and throws an exception if pointer is null. |
| if (!ptr)  throw std::invalid\_argument("null pointer");  \*ptr = 42; |

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

| **Principles(s):** 3 – Architect for Security & 9 – Effective QA: critical checks must run in release builds; tests prove they do. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GoogleTest | 1.14 | Unit test that fails if compile-time NDEBUG | removes logic checks. |
| clang-tidy | 18.1.1 | misc-static-assert | Warns when assert() guards essential behaviour. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Name: Do not throw from destructors during stack unwinding (ERR50-CPP)  Rationale: Throwing while another exception propagates terminates the program. |

| **Noncompliant Code** |
| --- |
| Destructor may throw if close() fails. |
| ~FileHandle() noexcept(false) {  if (close(fd) < 0)  throw std::runtime\_error("close failed");  } |

| **Compliant Code** |
| --- |
| Destructor is noexcept; logs errors instead of throwing. |
| ~FileHandle() noexcept {  if (close(fd) < 0)  std::fprintf(stderr, "close failed\n");  } |

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

| **Principles(s):** 4 – Keep It Simple & 8 – Defense in Depth: noexcept destructors guarantee predictable unwinding, avoiding hidden termination paths. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 18.1.1 | bugprone-exception-escape | Flags throws from destructors. |
| SonarQube | 2025.1 | cpp:S2229 | “Destructors should not throw exceptions”. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | STD-008-CPP | Name: Lock mutexes in consistent order (CON31-C)  Rationale: Inconsistent lock ordering creates deadlocks. |

| **Noncompliant Code** |
| --- |
| Two threads acquire the same mutexes in opposite order. |
| // Thread A  std::lock\_guard<std::mutex> L1(m1);  std::lock\_guard<std::mutex> L2(m2);  // Thread B  std::lock\_guard<std::mutex> L2(m2);  std::lock\_guard<std::mutex> L1(m1); // deadlock |

| **Compliant Code** |
| --- |
| Uses std::scoped\_lock to acquire both mutexes in a single consistent order. |
| std::scoped\_lock lock(m1, m2); // deadlock-free |

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

| **Principles(s):** 3 – Architect for Security & 8 – Defense in Depth: consistent lock order is a design-time rule that prevents runtime deadlock. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer (LLVM) | 18.1 | runtime data-race detector | Catches read/write races, lock misuse, and deadlocks while unit tests run |
| Helgrind (Valgrind) | 3.23 | lock-order analysis | Reports inconsistent mutex-acquisition order that can cause deadlock |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-009-CPP | Name: Use RAII to release resources on all paths (FIO46-C)  Rationale: Ensures files/sockets are closed even when exceptions occur. |

| **Noncompliant Code** |
| --- |
| Function leaks file descriptor when an early return happens. |
| FILE\* f = fopen(path, "r");  if (!f) return;  process(f);  if (error) return; // leak  fclose(f); |

| **Compliant Code** |
| --- |
| Uses std::ifstream; destructor closes automatically on every path. |
| std::ifstream file(path);  if (!file) return;  process(file); |

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

| **Principles(s):** 4 – Keep It Simple & 10 – Adopt Secure Coding Standard: RAII is the simplest, CERT-approved pattern to guarantee cleanup. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 18.1.1 | clang-analyzer-cplusplus.NewDeleteLeaks | Warns when resources aren’t released. |
| SonarQube | 2025.1 | cpp:S4742 | Detects missing close() / delete in exceptional paths. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging | STD-010-CPP | Name: Sanitize sensitive values before logging (FIO37-C)  Rationale: Logs are often archived; leaking secrets exposes users indefinitely. |

| **Noncompliant Code** |
| --- |
| Writes plaintext password to the application log. |
| log("Login failed: user=", username, " pwd=", password); |

| **Compliant Code** |
| --- |
| Logs the failure while omitting or hashing the password. |
| log("Login failed for user ", username, " [password omitted]"); |

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

| **Principles(s):** 7 – Sanitize Data Sent to Other Systems & 5 – Default Deny: treat logs like public data—scrub secrets unless explicitly whitelisted. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 2025.1 | cpp:S2637 | Flags logging of hard-coded credentials or variables named “password”. |
| Semgrep | 1.35 | Rule generic.log\_sensitive\_data | Greps for common secret-logging patterns during CI. |

### 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.

Green Pace folds security straight into the delivery pipeline by inserting a “Secure-Gate” stage between the Verify and Pre-production wedges of the DevSecOps wheel. Every push that hits the build server first runs static analysis with clang-tidy 18.1.1 and Cppcheck 2.5, then passes through a SonarQube 2025.1 quality gate; if either tool reports a blocker or critical issue, the build stops right there. Code that clears those checks is immediately fuzz-tested with LLVM libFuzzer to hammer new or changed functions with random and edge-case inputs, while unit tests execute under ThreadSanitizer for race detection and Valgrind for leak spotting. All findings land in a JSON report that a helper script feeds back into the master risk-assessment table, keeping severity and likelihood scores live and visible. Because the gate runs automatically on every branch and blocks merges when it spots trouble, security enforcement is continuous, hands-free, and perfectly aligned with the “verify and test to pre-production” flow shown in the diagram.

### 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 | Low | High | 5 |
| STD-003-CPP | High | Possible | Low | High | 4 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Possible | Medium | High | 4 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 3 |
| STD-007-CPP | High | Possible | Medium | High | 4 |
| STD-008-CPP | High | Possible | Medium | High | 4 |
| STD-009-CPP | Medium | Likely | Low | High | 4 |
| STD-010-CPP | High | Possible | Medium | High | 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.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | All persistent data, including DB files, backups, and log archives, must be encrypted with AES-256-GCM using customer-managed keys in AWS KMS. Keys rotate every 90 days and live only in an HSM. Applies to any store that holds PII or IP. |
| Encryption in flight | All traffic crossing a process boundary must use TLS 1.3 with forward secrecy (ECDHE suites). Mutual-TLS health checks block self-signed or expired certs during deployment. Covers internal microservice calls and all external endpoints. |
| Encryption in use | Sensitive fields (password hashes, bearer tokens, private keys) are kept in RAM encrypted via libsodium crypto\_secretbox() and wiped with explicit\_bzero() immediately after use. Protects against cold-boot and live-memory scraping. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All logins flow through OAuth 2.1 + PKCE. MFA is required for privileged roles. Access tokens are ES512-signed JWTs that expire after 15 minutes. |
| Authorization | Role-based access control (RBAC) tables define least-privilege scopes per microservice. Role changes require a pull request reviewed by SecOps; CI fails if a scope is added without docs. |
| Accounting | Every login, config change, or DB mutation writes an immutable JSON event to Elastic Search, chained with SHA-256 hashes. Daily jobs export digests to S3 Glacier for seven-year retention. |

**\***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

| **Stanard** | **Principle #(s)** | **Rationale (how the principle backs the rule)** |
| --- | --- | --- |
| STD-001-CPP | 2, 10 | Compiler warnings (2) catch mixed-sign maths; INT30-C is a CERT rule (10). |
| STD-002-CPP | 1, 3 | Input validation (1) and secure design (3) demand explicit bounds checks (ARR38-C). |
| STD-003-CPP | 1, 7 | Validate external strings (1) and sanitize outbound data (7) via safe APIs (STR03-C). |
| STD-004-CPP | 7, 5 | Sanitizing SQL (7) plus default-deny queries (5) prevents injection (FIO30-C). |
| STD-005-CPP | 8, 9 | Defense-in-depth (8) with runtime QA (9) pairs RAII and leak tools (MEM31-C). |
| STD-006-CPP | 3, 9 | Critical assertions stay live in release builds (3) and are unit-tested (9). |
| STD-007-CPP | 4, 8 | Simpler noexcept destructors (4) and layered error handling (8) avert unwinding chaos (ERR50-CPP). |
| STD-008-CPP | 3, 8 | Design-time lock order (3) plus runtime race checks (8) tame concurrency (CON31-C). |
| STD-009-CPP | 4, 10 | RAII is the simplest (4) CERT-endorsed cleanup pattern (10) (FIO46-C). |
| STD-0010-CPP | 7, 5 | Default-deny redaction (5) and sanitised log output (7) protect secrets (FIO37-C). |

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 |  |
| 1.2 | 05/29/2025 | Added milestone one | Jade Pineda |  |
| 1.5 | 06/15/2025 | Project one | Jade Pineda |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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