**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 | Validate Input Data stresses the importance of treating any data from external sources like it is potentially malicious. All input from the users, environmental variables, and network interfaces should be verified before use in production. Proper input validation can eliminate many software vulnerabilities and should be implemented for secure software development. |
| 1. Heed Compiler Warnings | Compiler warnings are an essential safeguard against insecure or unreliable code. Software developers should always compile their code with the highest warning level to view even the low-level compiler warnings. Steps should be taken to resolve these warnings through code modification rather than just ignoring them. This practice not only improves code correctness but also helps to eliminate potential vulnerabilities. Additional dynamic and static analysis tools should also be used to expose hidden security flaws and vulnerabilities. |
| 1. Architect and Design for Security Policies | Secure software begins with a software architecture that actively enforces security policies rather than relying on individual components that handle security later in the development cycle. For example, when a system requires different privilege levels at different times, the design should separate functionality into specific subsystems, each constrained to its minimal set of privileges. This approach reduces the attack surface and ensures that even if one subsystem is compromised, the breach does not automatically expose high level privileges of the system. |
| 1. Keep It Simple | Security is strengthened when systems are designed to be as simple as possible. Complex architectures increase the probability of implementation and configuration errors. Complex architectures also make the system harder to verify and maintain for developers and auditors. Assuring that your code is free from vulnerabilities grows more difficult with complex code bases and systems. By reducing complexity, developers can minimize the risk of hidden vulnerabilities and create systems that are easier to trust and secure. |
| 1. Default Deny | A secure system should assume that access is denied unless explicit permission is granted. Sometimes known as “whitelisting”, this principle ensures that protection mechanisms identify the exact conditions under which access is allowed rather than trying to block a seeming infinite range of disallowed actions. By basing access decisions on permission instead of exclusion, developers reduce the risk of accidental exposure caused by an overlooked case. |
| 1. Adhere to the Principle of Least Privilege | Each process in an application should operate with only the minimal set of privileges required to complete its task. Elevated permissions should be granted sparingly and only for the shortest duration necessary to complete the operation. This restriction significantly limits an attacker’s ability to exploit vulnerabilities and run arbitrary code with high level access. By carefully controlling privilege levels, software developers can reduce the potential damage of an exploited vulnerability. |
| 1. Sanitize Data Sent to Other Systems | Any data passed into complex subsystems (ex. Command shells, databases, or unused components in COTS (Commercial Off The Shelf) systems) must be thoroughly sanitized before use. Attackers often exploit these interfaces through injection attacks that invoke unintended functionality. Unlike input validation, sanitization ensures that the data is safe within the context of the specific subsystem being called. Since the invoking process understands this context, it should be responsible for properly sanitizing the data to prevent misuse in the subsystem. |
| 1. Practice Defense in Depth | Effective security relies on layering multiple defensive strategies so that no single point of failure compromises the system entirely. If one protective measure proves insufficient, another layer should block or mitigate the vulnerability, which reduces both the likelihood and the impact of a successful attack. For example, combining secure programming practices with hardened runtime environments creates overlapping safeguards that address vulnerabilities at different stages. |
| 1. Use Effective Quality Assurance Techniques | Strong quality assurance practices are important for identifying and mitigating vulnerabilities before deployment. Techniques such as fuzz testing, penetration testing, and source code audits can expose weaknesses that may not be revealed during normal development. Incorporating independent security reviews further enhances this process, as it allows an external reviewer to uncover invalid assumptions and overlooked security risks. |
| 1. Adopt a Secure Coding Standard | Creating secure software requires adherence to well defined coding standards that are designed for the target language and platform. A secure coding standard establishes consistent rules and best practices that directly address common sources of vulnerabilities (ex. Memory Management). By adopting and applying these standards on a software development team, teams can write more consistent code that builds maintainable and resilient systems. |

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-C++ | INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data |

| **Noncompliant Code** |
| --- |
| unsigned int a, b;  unsigned long c = a \* b; |
| Relies on unsigned long being later than unsigned int, which is not true on all platforms |

| **Compliant Code** |
| --- |
| uintmax\_t c = a \* b; |
| Uses uintmax\_t to ensure the result fits regardless of platform |

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

| **Principles(s):** **Heed Compiler Warnings –** Unsafe integer conversions are frequently caught first by compiler diagnostics. Developers that eliminate these warnings prevent overflow or truncation bugs that lead to memory corruption. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | N/A | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| CodeSonar | 9.1p0 | LANG.CAST.PC.AV LANG.CAST.PC.CONST2PTR LANG.CAST.PC.INT  LANG.CAST.COERCE LANG.CAST.VALUE  ALLOC.SIZE.TRUNC MISC.MEM.SIZE.TRUNC  LANG.MEM.TBA | Flags value-altering casts and size truncation patterns tied to unsafe conversion. |
| Helix QAC | 2025.2 | C2850, C2855, C2890, C2895, C2900, C2905,  C++2850, C++2855, C++2890, C++2895, C++2900, C++2905,  C++3000, C++3010  DF2851, DF2852, DF2853,  DF2856, DF2857, DF2858, DF2891, DF2892, DF2893, DF2896, DF2897, DF2898, DF2901, DF2902, DF2903, DF2906, DF2907, DF2908 | Broad rule set convering integer conversion issues and truncation in expressions and allocation sizes. |
| Parasoft C/C++test | 2024.2 | |  |  | | --- | --- | |  | **CERT\_C-INT31-a CERT\_C-INT31-b CERT\_C-INT31-c CERT\_C-INT31-d CERT\_C-INT31-e CERT\_C-INT31-f CERT\_C-INT31-g CERT\_C-INT31-h CERT\_C-INT31-i CERT\_C-INT31-j CERT\_C-INT31-k CERT\_C-INT31-l CERT\_C-INT31-m CERT\_C-INT31-n CERT\_C-INT31-o CERT\_C-INT31-p** | | Dedicated CERT INT31 checks including range validation, signed/unsigned conversions, Boolean/enum misuse in arithmetic and more. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-C++ | Validate and Sanitize Data Values |

| **Noncompliant Code** |
| --- |
| #include <iostream>  void runCommand(const std::string& userInput) {  system(userInput.c\_str());  } |
| Uses unsanitized input in a system call, allowing the injection of arbitrary commands |

| **Compliant Code** |
| --- |
| #include <iostream>  void runCommand(const std::string& userInput) {  if (userInput == “show files”) {  system(“ls”); // Only allow safe, predefined commands  } else {  std::cout << “Invalid command.”;  }  } |
| Restricts values to known safe inputs, preventing command injection. Does not allow direct user input for function calls. |

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

| **Principles(s):** **Validate Input Data –** Reading or operating on unsanitized values yields indeterminate behavior that can further cause memory corruption. Developers should initialize every object before use and reject out of range data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Warns on potential use before initialization and other undefined results |
| CodeSonar | 9.1p0 | |  | | --- | | **LANG.STRUCT.RPL** **LANG.MEM.UVAR** | | Flags uninitialized variable usage and returning pointers to locals |
| Klocwork | 2025.2 | UNINIT.CTOR.MIGHT UNINIT.CTOR.MUST UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST UNINIT.STACK.ARRAY.MIGHT UNINIT.STACK.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST UNINIT.STACK.MIGHT UNINIT.STACK.MUST | Detects uninitialized reads across stacks, heaps, arrays, and constructors |
| Parasoft C/C++ test | 2024.2 | CERT\_CPP-EXP53-a | Checks for “Avoid use before initialization” |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-C++ | STR51-CPP. Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| #include <cstdlib>  #include <string>    void f() {    std::string tmp(std::getenv("TMP"));    if (!tmp.empty()) {      // ...    }  } |
| A std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs).   |  | | --- | |  | |

| **Compliant Code** |
| --- |
| #include <cstdlib>  #include <string>    void f() {    const char \*tmpPtrVal = std::getenv("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");    if (!tmp.empty()) {      // ...    }  } |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |

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

| **Principles(s):** **Validate Input Data –** Passing a null pointer into string constructors or operations ultimately can dereference it, which can result in undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | |  |  | | --- | --- | |  | **assert\_failure** | | Flags paths that trigger assertions, including null dereference in string operations. |
| CodeSonar | 9.1p0 | LANG.MEM.NPD | Detects potential null pointer dereferences when calling std::string APIs with null C-strings. |
| Helix QAC | 2025.2 | DF4770, DF4771, DF4772, DF4773, DF4774 | Diagnostics covering null checks and dangerous string operations that can lead to dereference. |
| Klocwork | 2025.2 | NPD.CHECK.CALL.MIGHT NPD.CHECK.CALL.MUST NPD.CHECK.MIGHT NPD.CHECK.MUST NPD.CONST.CALL NPD.CONST.DEREF NPD.FUNC.CALL.MIGHT NPD.FUNC.CALL.MUST NPD.FUNC.MIGHT NPD.FUNC.MUST NPD.GEN.CALL.MIGHT NPD.GEN.CALL.MUST NPD.GEN.MIGHT NPD.GEN.MUST RNPD.CALL RNPD.DEREF | Rules that track null flow into function calls and dereferences |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-C++ | IDS00-J. Prevent SQL injection |

| **Noncompliant Code** |
| --- |
| #include <string> #include <sqlite3>  int export\_sorted(sqlite3\* db, const std::string& sortBy) {  std::string sql = “SELECT id, username FROM users ORDER BY ” + sortBy;  char\* err = nullptr;  return sqlite3\_exec(db, sql.c\_str(), nullptr, nullptr, &err)  } |
| Identifiers cannot be parameterized and concatenating them enables injection through crafted input. |

| **Compliant Code** |
| --- |
| #include <string>  #include <unordered\_set> #include <sqlite3>  int export\_sorted(sqlite3\* db, const std::string& sortBy) {  static const std::unordered\_set<std::string> allowed = {"id", "username", "created\_at"};  const std::string key = allowed.count(sortBy) ? sortBy : "id"; // whitelist  const std::string sql = "SELECT id, username FROM users ORDER BY " + key;  char\* err = nullptr;  return sqlite3\_exec(db, sql.c\_str(), nullptr, nullptr, &err);  } |
| Whitelist the set of allowed identifiers to search the database, preventing attacker controlled SQL syntax. |

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

| **Principles(s):** **Validate Input Data –** Unvalidated or unsanitized inputs can alter query logic. By validating the input data before it reaches a database command, developers ensure that only expected input is processed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | JAVA.IO.INJ.SQL | SQL injection |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| FindBugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Klocwork | 2025.2 | SV.DATA.DB  SV.SQL  SV.SQL.DBSOURCE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-C++ | MEM50-CPP. Do not access freed memory |

| **Noncompliant Code** |
| --- |
| struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |
| s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. |

| **Compliant Code** |
| --- |
| struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |
| Dynamically allocated memory is not deallocated until it is no longer required. |

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

| **Principles(s):** **Keep it simple –** Manual memory management often can lead to data leaks and use-after-free errors. Developers can simplify memory ownership through modern C++ tools like smart pointers, and with the principle of RAII (Resource Acquisition Is Initialization) where a resource is acquired in an object’s constructor and released in its destructor. This keeps memory handling simple and explicit to avoid undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use | N/A |
| Coverity | V7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Parasoft C/C++ test | 2024.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Klocwork | 2025.2 | UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST | N/A |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-C++ | MSC11-C. Incorporate diagnostic tests using assertions |

| **Noncompliant Code** |
| --- |
| #include <cassert>  int divide(int a, int b) {  assert(b !=0);  return a / b;  } |
| Code relies on an assertion to prevent division by zero, which can fail silently in production builds. |

| **Compliant Code** |
| --- |
| #include <cassert>  int divide(int a, int b) {  if (b == 0) {  throw std::runtime\_error(“Division by zero”);  }  Return a / b;  } |
| Checks for invalid input explicitly and raises an exception, preserving program safety regardless of build type. |

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

| **Principles(s):** **Design for Security –** Assertions are used for checking internal assumptions during the development process, not for validating runtime conditions. Assert() is typically disabled in production removing the protection that the developer intended to implement. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | |  |  | | --- | --- | | **LANG.FUNCS.ASSERTS** |  | | Not enough assertions |
| Coverity | 2017.07 | |  |  | | --- | --- | | **ASSERT\_SIDE\_EFFECT** |  | | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++ test | 2024.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |
| Security Reviewer – Static Reviewer | 6.02 | CPPPBE | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-C++ | ERR56-CPP. Guarantee Exception Safety |

| **Noncompliant Code** |
| --- |
| #include <stdexception>  void test\_func() {  try {  throw std::runtime\_error(“fail”);  } catch (std::exception e) {  // …  throw e;  }  } |
| Catches by value and rethrows with throw e; which discards the original exception context in the stack trace |

| **Compliant Code** |
| --- |
| #include <stdexception>  void test\_func() {  try {  throw std::runtime\_error(“fail”);  } catch (const std::exception& e) {  // …  throw;  }  } |
| Catches by const reference and rethrows with throw;, preserving type and diagnostics in the stackt race. |

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

| **Principles(s):** **Use Effective Quality Assurance Techniques –** Exception safety requires designing and testing code that throws exceptions to ensure resources don’t leak. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2025.2 | C++4075, C++4076 | N/A |
| LDRA tool suite | 9.7.1 | 527 S, 56 D, 71 D | Always catch exceptions Empty 'catch' blocks should not be used |
| Polyspace Bug Finder | R2025b | CERT C++: ERR56-CPP | Checks for exceptions violating class invariant (rule fully covered). |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency & Thread Safety | STD-008-C++ | CON52-CPP. Prevent data races when accessing bit-fields from multiple threads |

| **Noncompliant Code** |
| --- |
| #include <thread>  #include <vector>  int counter = 0;  void inc() {  for (int i = 0; i < 100000; ++i) counter++;  }  void main() {  std::vector<std::thread> ts(4);  for (auto& t : ts) t = std::thread(inc);  for (auto& t : ts) t.join();  } |
| Multiple threads write the same variable without synchronization, causing a data race and undefined behavior. |

| **Compliant Code** |
| --- |
| #include <thread>  #include <vector>  #include <atomic>  std::atomic<int> counter{0};  void inc() {  for (int i = 0; i < 100000; ++i) counter.fetch\_add(1, std::memory\_order\_relaxed);  }  void main() {  std::vector<std::thread> ts(4);  for (auto& t : ts) t = std::thread(inc);  for (auto& t : ts) t.join();  } |
| Replaces unsynchronized writes with atomic increments, eliminating the race. |

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

| **Principles(s):** **Architect and design for security policies –** Developers should plan thread interactions so that every access to shared states are protected and data layouts avoid unintended sharing. Well defined synchronization policies prevent data races and undefined behavior in concurrent code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | read\_write\_data\_race  write\_write\_data\_race | Supported |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-CON52 | N/A |
| CodeSonar | 9.1p0 | CONCURRENCY.DATARACE | Data Race |
| Coverity | 6.5 | RACE\_CONDITION | Fully Implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Randomness | STD-009-C++ | MSC51-CPP. Ensure your random number generator is properly seeded |

| **Noncompliant Code** |
| --- |
| #include <random>  int weak\_random() {  std::mt19937 gen(12345);  return gen() % 100;  } |
| A deterministic seed makes output reproducible, and modulo skews the distribution of randomness. |

| **Compliant Code** |
| --- |
| #include <random>  int strong\_random() {  std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<int> dist(0,99);  return dist(gen);  } |
| Seeds the engine from an entropy source and uses a proper distribution, eliminating predictability and reproducibility. |

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

| **Principles(s):** **Architect and design for security policies** **–** Random Number Generator choice and seeding for this generator must be considered part of the software architecture. Developers should always use cryptographically sound generators, and these generators should be seeded with entropy so that an attacker cannot predict or control the number generation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | default-construction | Partially Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC51 | N/A |
| CodeSonar | 9.1p0 | HARDCODED.SEED  MISC.CRYPTO.TIMESEED | Hardcoded Seed in PRNG  Predictable Seed in PRNG |
| Parasoft C/C++ test | 2024.2 | |  |  | | --- | --- | |  | **CERT\_CPP-MSC51-a** | | Properly seed pseudorandom number generators |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secrets Management | STD-010-C++ | MSC41-C. Never hard code sensitive information |

| **Noncompliant Code** |
| --- |
| #include <iostream>  #include <string>  std::string connect\_str() {  const std::string user = “admin”;  const std::string pass = “secretpassword123”;  std::cout << “Connecting to server with user=” << user << “ password=” << pass << \n;  return “user=” + user + “|password=” + pass;  } |
| Embeds the password in the binary and prints it to logs, making exposure likely |

| **Compliant Code** |
| --- |
| #include <string>  #include <stdexcept>  std::string connect\_str() {  const char\* pw = std::getenv(“DB\_PASSWORD”);  if (!pw || !\*pw) throw std::runtime\_error(“Missing DB\_PASSWORD”);  // Do not log the password  return std::string(“user=app|password=”) + pw;  } |
| Obtains the secret at runtime via an environment variable and avoids logging it, reducing exposure in the binary and in repos |

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

| **Principles(s):** **Adhere to the Principle of Least Privilege –** Sensitive information like passwords, API tokens, and encryption keys should never be exposed to code or to users who don’t strictly need access. Hard-coding secrets violates least-privilege design by granting every build of the software direct access to credentials that should be kept isolated. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | N/A | Supported |
| CodeSonar | 9.1p0 | HARDCODED.AUTH  HARDCODED.DNS  HARDCODED.KEY  HARDCODED.SALT  HARDCODED.SEED | Hardcoded Authentication  Hardcoded DNS Name  Hardcoded Crypto Key  Hardcoded Crypto Salt  Hardcoded Seed in PRNG |
| Helix QAC | 2025.2 | |  |  | | --- | --- | |  | DF3556, DF3557, DF3558  C++3842 | |  |  | | N/A |
| Parasoft C/C++test | 2024.2 | CERT\_C-MSC41-a | Do not hard code string literals |

### 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 integrated into our DevSecOps workflow to continuously enforce the coding standards we define in this policy. In the **Pre-Production** phase, automation can be introduced during the **Build** and **Verify and Test** stages by integrating static analysis tools, unit tests, and vulnerability scanning into the development environment. These automated steps ensure that code is validated against the SEI CERT standards while it is being written. In the **Production** phase, automated deployment scripts can apply config hardening and initiate regression/penetration testing in the **Transition and Health Check** stage. The **Monitor and Detect** stage also implement automation by incorporating event logging and monitoring to alert developers when security policies have been violated.

### 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 | Probable | Medium | P12 | L1 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P9 | L2 |
| STD-006-CPP | Low | Unlikely | Low | P1 | L3 |
| STD-007-CPP | High | Likely | Medium | P9 | L2 |
| STD-008-CPP | Medium | Probable | Medium | P4 | L3 |
| STD-009-CPP | Medium | Likely | Medium | P18 | L1 |
| STD-010-CPP | High | Probable | Medium | P6 | L2 |

### 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 | Encryption at rest protects data stored on physical storage media like hard drives, SSDs, USB drives, etc. Green Pace should require that all sensitive data and PII (Personally Identifiable Information) be encrypted using industry standard algorithms like AES-256 to protect data in the event of a physical data breach on premises. |
| Encryption in flight | Encryption in flight protects data as it travels across networks. Green Pace should require the use of secure communication protocols like TLS to prevent man-in-the-middle attacks. All Green Pace APIs, web applications, and remote connections to their network should enforce HTTPS or encrypted VPN connections. |
| Encryption in use | Encryption in use safeguards data that is actively processed in the memory stack of an application. Green Pace should enforce this through application-level encryption that protect sensitive variables during runtime. This prevents the exposure of confidential data through memory dumps or reverse engineering to ensure that data is not leaked outside of its application space. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identify of users accessing Green Pace’s resources, services, and APIs. All user logins must require multifactor authentication to ensure only authorized users gain access to these systems. |
| Authorization | Authorization governs which authenticated users are permitted to do what once access has been granted. RBAC (Role-based access control) is required to restrict permissions based on employee type. This ensures users can only modify data relevant to their responsibilities. Changes to user roles and access levels must be regularly reviewed and modified as required. |
| Accounting | Accounting ensures that all user actions are logged, monitored, and available to be audited. This includes tracking logins, file access, config changes, and database changes. Audit logs should be time-stamped, secured, and regularly reviewed by developers to detect policy violations. |

### 

### 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

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 | 10/16/2025 | Project One | Connor Bailey | Connor Bailey |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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