**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 c, 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 treat input as untrusted. Validate all input data rigorously using whitelisting, proper type checks, and length constraints to prevent injection attacks, buffer overflows, and other vulnerabilities. |
| 1. Heed Compiler Warnings | Modern compilers offer valuable warnings that can help identify potential issues early in development. Enabling the strictest warning levels and treating warnings as errors ensures cleaner, safer code. |
| 1. Architect and Design for Security Policies | Security must be built into the architecture from the start. Design applications with secure authentication, access control, and data protection policies that align with organizational standards. |
| 1. Keep It Simple | Simpler code is easier to audit and less prone to errors. Avoid unnecessary complexity and abstractions that can obscure logic and introduce subtle bugs or security flaws. |
| 1. Default Deny | Deny access by default and only allow explicitly authorized actions. This reduces the attack surface and ensures that unintended access paths are not exposed. |
| 1. Adhere to the Principle of Least Privilege | Code and users should only have the permissions necessary to perform their tasks. Limiting privileges helps contain damage in the event of a breach or software flaw. |
| 1. Sanitize Data Sent to Other Systems | Before outputting data to other systems (e.g., databases, browsers, file systems), sanitize it to prevent injection attacks and ensure compatibility and integrity. |
| 1. Practice Defense in Depth | Implement multiple layers of defense to protect against failures at any single point. This includes firewalls, encryption, access control, and runtime checks, creating a resilient security posture. |
| 1. Use Effective Quality Assurance Techniques | Secure software development relies on thorough testing, including unit, integration, and fuzz testing, to detect bugs and vulnerabilities early and ensure code reliability. |
| 1. Adopt a Secure Coding Standard | Use established coding standards such as CERT C/C++ to enforce consistency and eliminate common security flaws. These standards provide guidance on secure practices and prevent recurring issues. |

### 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** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Improper conversions between signed and unsigned integers or different widths can result in data loss or unintended behavior, introducing security risks and data integrity issues. |

| **Noncompliant Code** |
| --- |
| This code casts a signed int to an unsigned int without checking the sign, which can result in negative values being interpreted as large positive numbers. |
| #include <limits.h>  void func(signed int si) {  /\* Cast eliminates warning \*/  unsigned int ui = (unsigned int)si;  /\* ... \*/  }  /\* ... \*/  func(INT\_MIN); |

| **Compliant Code** |
| --- |
| This version validates the value of the signed integer before casting it to an unsigned type to avoid misinterpretation. |
| #include <limits.h>  void func(signed int si) {  unsigned int ui;  if (si < 0) {  /\* Handle error \*/  } else {  ui = (unsigned int)si; /\* Cast eliminates warning \*/  }  /\* ... \*/  }  /\* ... \*/  func(INT\_MIN + 1); |

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

| **Principles(s):**  1 – Validate Input Data: Prevents unexpected behavior from improper conversions.  10 – Adopt a Secure Coding Standard: Using SEI CERT INT31-CPP avoids conversion vulnerabilities.  4 – Keep It Simple: Using clear validation logic reduces ambiguity and error-prone complexity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | [-Wconversion](https://clang.llvm.org/docs/DiagnosticsReference.html#wconversion), [-Wsign-conversion](https://clang.llvm.org/docs/DiagnosticsReference.html#wsign-conversion) | Detects implicit type conversions that may result in lost or misinterpreted data. |
| SonarQube | 4.10 | [cpp:S881](https://rules.sonarsource.com/cpp/RSPEC-881/) | Flags unsafe comparisons between signed and unsigned integers. |
| Polyspace Bug Finder | R2024b | [CERT C: Rule INT31-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint31c.html) | Detects violations of SEI CERT rule INT31-C for integer conversion safety. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not dereference null pointers** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Dereferencing null pointers causes undefined behavior and can result in application crashes or security vulnerabilities. This rule ensures pointers are validated before use to prevent such critical failures. |

| **Noncompliant Code** |
| --- |
| This noncompliant code fails to validate the result of png\_malloc(). If memory allocation fails, the memcpy() function attempts to write to a null pointer, leading to undefined behavior. |
| #include <png.h> /\* From libpng \*/  #include <string.h>    void func(png\_structp png\_ptr, int length, const void \*user\_data) {  png\_charp chunkdata;  chunkdata = (png\_charp)png\_malloc(png\_ptr, length + 1);  /\* ... \*/  memcpy(chunkdata, user\_data, length);  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant code validates that the length, user\_data, and the result of png\_malloc() are not null before calling memcpy(). It uses size\_t for safer arithmetic. |
| #include <png.h> /\* From libpng \*/  #include <string.h>  void func(png\_structp png\_ptr, size\_t length, const void \*user\_data) {  png\_charp chunkdata;  if (length == SIZE\_MAX) {  /\* Handle error \*/  }  if (NULL == user\_data) {  /\* Handle error \*/  }  chunkdata = (png\_charp)png\_malloc(png\_ptr, length + 1);  if (NULL == chunkdata) {  /\* Handle error \*/  }  /\* ... \*/  memcpy(chunkdata, user\_data, length);  /\* ... \*/  } |

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

| **Principles(s):**  1 - Validate Input Data – Ensures memory allocation and user input are valid.  8 - Practice Defense in Depth – Adds checks at multiple points to prevent null dereference.  10 - Adopt a Secure Coding Standard – Follows SEI CERT EXP33-CPP. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | [-Wnull-dereference](https://clang.llvm.org/docs/DiagnosticsReference.html#wnull-dereference) | Detects execution paths where a null pointer may be dereferenced. |
| Cppcheck | 2.12.0 | nullPointer | Identifies unsafe pointer dereferences at compile time. |
| SonarQube | 4.10 | [cpp:S2259](https://rules.sonarsource.com/cpp/RSPEC-2259/) | Detects conditions where null pointers may be dereferenced. |
| Polyspace Bug Finder | R2024b | [CERT C: Rule EXP33-C](https://www.mathworks.com/help/bugfinder/ref/certcruleexp33c.html) | Flags null pointer dereference violations per SEI CERT EXP33-CPP. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Failing to allocate sufficient memory for strings can cause buffer overflows and corrupt memory, leading to exploitable vulnerabilities. This rule ensures that all string operations leave space for the null terminator and prevent overruns. |

| **Noncompliant Code** |
| --- |
| This noncompliant example uses sprintf() without bounding the %s conversion, risking a buffer overflow if name exceeds the size of the filename buffer. |
| #include <stdio.h>    void func(const char \*name) {  char filename[128];  sprintf(filename, "%s.txt", name);  } |

| **Compliant Code** |
| --- |
| This compliant version uses a precision specifier to ensure the string copy fits safely within the buffer, including the file extension and null terminator. |
| #include <stdio.h>    void func(const char \*name) {  char filename[128];  sprintf(filename, "%.123s.txt", name);  } |

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

| **Principles(s)**  1 – Validate Input Data: Input string length is bounded to avoid overflow.  4 – Keep It Simple: Direct bounds enforcement keeps the logic readable and safe.  10 – Adopt a Secure Coding Standard: Follows best practices for safe formatting (CERT STR50-CPP). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | [-Wformat-overflow](https://clang.llvm.org/docs/DiagnosticsReference.html#wformat-overflow) | Warns when sprintf or related functions may exceed buffer size. |
| Cppcheck | 2.12.0 | bufferOverrun, sprintfSize | Detects risky string formatting and buffer size mismatches. |
| Polyspace Bug Finder | R2024b | [CERT C: Rule STR31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr31c.html) | Flags improper use of string formatting functions violating null-termination rules. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Untrusted user input used as format strings can lead to format string vulnerabilities, including reading arbitrary memory and remote code execution. Format functions must use constant format strings and pass user input as separate arguments. |

| **Noncompliant Code** |
| --- |
| This example uses snprintf() with user-controlled format data, then passes it to syslog()—a pattern vulnerable to format string attacks. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <syslog.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  syslog(LOG\_INFO, msg);  free(msg);  } |

| **Compliant Code** |
| --- |
| This compliant version ensures that the untrusted user input is only passed as an argument to fprintf(), not as the format string itself. |
| #include <stdio.h>    void incorrect\_password(const char \*user) {  static const char msg\_format[] = "%s cannot be authenticated.\n";  fprintf(stderr, msg\_format, user);  } |

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

| **Principles(s):**  1 – Validate Input Data: Ensures untrusted strings are treated as data, not executable instructions.  7 – Sanitize Data Sent to Other Systems: Prevents user input from being misinterpreted in external APIs.  10 – Adopt a Secure Coding Standard: Aligns with CERT STR11-CPP and FIO30-C for format string safety. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | [-Wformat-security](https://clang.llvm.org/docs/DiagnosticsReference.html#wformat-security) | Warns if format string in a function like printf() is not a string literal. |
| Cppcheck | 2.12.0 | formatString | Detects format string vulnerabilities including non-literal format use. |
| Polyspace Bug Finder | R2024b | [CERT C: Rule FIO30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulefio30c.html) | Detects format string injection vulnerabilities due to untrusted format input. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Mixing incompatible memory management functions like malloc() with delete, or new[] with delete instead of delete[], leads to undefined behavior, memory corruption, and potential crashes. This rule ensures proper pairing of allocation and deallocation functions to maintain program integrity. |

| **Noncompliant Code** |
| --- |
| This example allocates memory with malloc() but incorrectly deallocates it using delete. |
| #include <cstdlib>  void f() {  int \*i = static\_cast<int \*>(std::malloc(sizeof(int)));  // ...  delete i;  } |

| **Compliant Code** |
| --- |
| This compliant solution deallocates memory using the corresponding free() function after allocating with malloc(). |
| #include <cstdlib>  void f() {  int \*i = static\_cast<int \*>(std::malloc(sizeof(int)));  // ...  std::free(i);  } |

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

| **Principles(s):**  5 – Default Deny: Memory cleanup reduces the attack surface and avoids dangling references.  9 – Use Effective Quality Assurance Techniques: Mismatched memory routines can be caught with tools and testing.  10 – Adopt a Secure Coding Standard: CERT MEM51-CPP requires correct allocation-deallocation pairing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | unix.Malloc, c++:NewDeleteLeaks | Detects mismatched allocation/deallocation and memory leaks. |
| Cppcheck | 2.12.0 | memleak, mismatchingAllocationDeallocation | Identifies mismatched new/delete, malloc/free use. |
| Polyspace Bug Finder | R2024b | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Detects mismatched memory management routines in accordance with MEM51-CPP. |
| Valgrind | 3.21.0 | [memcheck](https://valgrind.org/docs/manual/mc-manual.html) | Runtime tool that detects invalid frees and leaks during execution. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | If exceptions are not caught by a matching handler, the C++ runtime calls std::terminate(), which may bypass cleanup code and leave resources unreleased. Catching all exceptions ensures proper resource management and prevents denial-of-service via abnormal termination. |

| **Noncompliant Code** |
| --- |
| This code launches a thread that executes a function which throws, but the exception is not caught, causing the program to terminate abnormally. |
| #include <thread>  void throwing\_func() noexcept(false);    void thread\_start() {  throwing\_func();  }    void f() {  std::thread t(thread\_start);  t.join();  } |

| **Compliant Code** |
| --- |
| This compliant solution catches all exceptions thrown in the thread function to ensure controlled termination. |
| #include <thread>  void throwing\_func() noexcept(false);  void thread\_start(void) {  try {  throwing\_func();  } catch (...) {  // Handle error  }  }  void f() {  std::thread t(thread\_start);  t.join();  } |

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

| **Principles(s):**  8 – Practice Defense in Depth: Provides fault isolation through exception handling layers.  9 – Use Effective Quality Assurance Techniques: Ensures exception paths are tested and covered.  10 – Adopt a Secure Coding Standard: Follows CERT ERR51-CPP and ensures handlers are present for thrown exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | c++:UncaughtExceptions | Warns if exceptions may escape without being caught. |
| Cppcheck | 2.12.0 | uncaughtException | Detects functions or threads where exceptions may not be caught. |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR51-a | Detects unhandled exceptions per CERT ERR51-CPP. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Do not let exceptions escape from destructors or deallocation functions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Throwing exceptions from a destructor or deallocation function during stack unwinding results in undefined behavior and typically leads to program termination via std::terminate. Such conditions prevent resource cleanup and violate program reliability guarantees. |

| **Noncompliant Code** |
| --- |
| This noncompliant code throws an exception from a custom global deallocation function, which can trigger undefined behavior. |
| #include <stdexcept>    bool perform\_dealloc(void \*);    void operator delete(void \*ptr) noexcept(false) {  if (perform\_dealloc(ptr)) {  throw std::logic\_error("Something bad");  }  } |

| **Compliant Code** |
| --- |
| The compliant solution logs the error and terminates the program without throwing, ensuring no exceptions escape during deallocation. |
| #include <cstdlib>  #include <stdexcept>    bool perform\_dealloc(void \*);  void log\_failure(const char \*);    void operator delete(void \*ptr) noexcept(true) {  if (perform\_dealloc(ptr)) {  log\_failure("Deallocation of pointer failed");  std::exit(1); // Fail, but still call destructors  }  } |

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

| **Principles(s):**  8 – Practice Defense in Depth: Avoids cascading failure during error handling.  9 – Use Effective Quality Assurance Techniques: Prevents hidden runtime failures due to exception propagation.  10 – Adopt a Secure Coding Standard: Ensures adherence to ERR52-CPP. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | [-Wexceptions](https://clang.llvm.org/docs/DiagnosticsReference.html#wexceptions), noexcept-related analysis | Warns when destructors marked noexcept(false) may cause termination. |
| Cppcheck | 2.12.0 | throwInDestructor | Detects when exceptions may escape destructors or delete functions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Guarantee that container indices and iterators are within the valid range** |
| --- | --- | --- |
| **Container Safety** | STD-008-CPP | Using invalid indices or iterators to access elements in containers like arrays, vectors, or strings causes undefined behavior. This can corrupt memory, trigger access violations, or crash programs. Always verify indices and iterators before use. |

| **Noncompliant Code** |
| --- |
| This code assumes that the iterator b is valid without confirming that it precedes e. If the container is empty, b == e, and dereferencing b results in undefined behavior. |
| #include <iterator>    template <typename ForwardIterator>  void f\_imp(ForwardIterator b, ForwardIterator e, int val, std::forward\_iterator\_tag) {  do {  \*b++ = val;  } while (b != e);  }  template <typename ForwardIterator>  void f(ForwardIterator b, ForwardIterator e, int val) {  typename std::iterator\_traits<ForwardIterator>::iterator\_category cat;  f\_imp(b, e, val, cat);  } |

| **Compliant Code** |
| --- |
| This solution ensures iterator b is not equal to e before it is dereferenced. |
| #include <iterator>    template <typename ForwardIterator>  void f\_imp(ForwardIterator b, ForwardIterator e, int val, std::forward\_iterator\_tag) {  while (b != e) {  \*b++ = val;  }  }  template <typename ForwardIterator>  void f(ForwardIterator b, ForwardIterator e, int val) {  typename std::iterator\_traits<ForwardIterator>::iterator\_category cat;  f\_imp(b, e, val, cat);  } |

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

| **Principles(s):**  1 – Validate Input Data: Ensures iterator boundaries are respected.  9 – Use Effective Quality Assurance Techniques: Iteration logic is verified with runtime and static checks.  10 – Adopt a Secure Coding Standard: Aligns with CERT rules for iterator safety, like CTR51-CPP and CTR53-CPP. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | [-Warray-bounds](https://clang.llvm.org/docs/DiagnosticsReference.html#warray-bounds), -Witerator-range | Detects out-of-bounds container access and iterator misuse. |
| Cppcheck | 2.12.0 | outOfBounds, invalidIterator | Warns on potential index or iterator range violations. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR53-a, CERT\_CPP-CTR51-a | Identifies misuse of STL containers and invalid iterator access. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Range check element access** |
| --- | --- | --- |
| **String Access** | STD-009-CPP | Accessing string elements without verifying the bounds can lead to undefined behavior and security vulnerabilities. Validating string indices prevents out-of-bound access, which may cause memory corruption or crashes. |

| **Noncompliant Code** |
| --- |
| This code attempts to modify the first character of a string without checking if the string is empty, which causes undefined behavior when the string is empty. |
| #include <string>  #include <locale>  void capitalize(std::string &s) {  std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

| **Compliant Code** |
| --- |
| This code ensures the string is non-empty before accessing the first element. |
| #include <string>  #include <locale>  void capitalize(std::string &s) {  if (s.empty()) {  return;  }  std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

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

| **Principles(s):**  1 – Validate Input Data: Confirms the string is safe to access.  9 – Use Effective Quality Assurance Techniques: Promotes detection of boundary violations through runtime and static checks.  10 – Adopt a Secure Coding Standard: Aligns with STR51-CPP and STR53-CPP for safe string access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | -Wstring-bounds, [-Wtautological-compare](https://clang.llvm.org/docs/DiagnosticsReference.html#wtautological-compare) | Detects invalid string access and always-false comparisons. |
| Cppcheck | 2.12.0 | outOfBounds, stringIndexError | Detects unsafe string indexing and access to potentially empty strings. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a, CERT\_CPP-STR53-a | Ensures bounds checks are present for string and array element access. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Obey the one-definition rule** |
| --- | --- | --- |
| **Declarations** | STD-010-CPP | Violating the One-Definition Rule (ODR) causes undefined behavior, particularly when conflicting types or definitions are compiled into separate translation units. Such inconsistencies lead to memory corruption, unexpected results, or compiler/linker errors. |

| **Noncompliant Code** |
| --- |
| This noncompliant example includes the same header in multiple translation units (a.cpp, s.cpp) but with differing alignment due to #pragma pack, violating ODR. |
| // s.h  struct S {  char c;  int a;  };    void init\_s(S &s);    // s.cpp  #include "s.h"    void init\_s(S &s); {  s.c = 'a';  s.a = 12;  }    // a.cpp  #pragma pack(push, 1)  #include "s.h"  #pragma pack(pop)    void f() {  S s;  init\_s(s);  } |

| **Compliant Code** |
| --- |
| In this compliant version, class definitions are unique across translation units, preventing any layout conflicts or ODR violations. |
| // a.cpp  namespace {  struct S {  int a;  };  }    // b.cpp  namespace {  class S {  public:  int a;  };  } |

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

| **Principles(s):**  3 – Architect and Design for Security Policies: Enforces modular consistency in type design.  4 – Keep It Simple: Promotes consistent, centralized declarations to avoid hard-to-detect bugs.  10 – Adopt a Secure Coding Standard: Aligns with DCL60-CPP for unique and consistent type definitions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 17 | [-Wodr](https://clang.llvm.org/docs/DiagnosticsReference.html#wodr) | Warns when multiple definitions of the same symbol conflict across translation units. |
| Cppcheck | 2.12.0 | duplicateSymbol, odrViolation | Detects One-Definition Rule violations across separate translation units. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL60-a | Detects ODR violations and conflicting declarations as per SEI CERT guidance. |

### 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 will be used to enforce compliance with the secure coding standards defined in this policy. Green Pace’s existing DevOps process will be extended to a DevSecOps model by integrating automated security tools throughout the software development lifecycle. In the Design and Build stages, static analysis tools such as Clang-Tidy, Cppcheck, and Parasoft C/C++test will be embedded into development environments and CI pipelines to detect violations like unsafe memory usage, improper exception handling, and unsafe string formatting. These tools will automatically flag noncompliant code and prevent builds from proceeding until all policy violations are resolved.

During the Verify and Test phase, memory analysis tools like Valgrind will be used to catch issues such as leaks or invalid frees. Automation ensures consistent enforcement of critical rules, including those related to the One-Definition Rule, proper iterator use, and null pointer dereferencing. In later phases like Monitor and Detect, runtime behaviors will be logged and monitored to identify violations that might escape static analysis. By integrating security checks at every phase — from planning through post-deployment — Green Pace can ensure that policy compliance is continuous, measurable, and enforceable as part of its defense-in-depth strategy.

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

### 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 stored data by making it unreadable without proper authorization. This includes hard drives, cloud storage, databases, and backup systems. The policy ensures that all sensitive data (such as credentials, financial records, and personal information) is encrypted using AES-256 or equivalent before it is written to disk. This reduces the risk of data theft if storage is compromised. |
| Encryption in flight | Encryption in flight protects data as it moves across networks, using protocols such as TLS 1.3. This policy mandates that all internal and external communications (API calls, database queries, login credentials) be encrypted to prevent interception or tampering. It applies to both public internet and internal service communications. |
| Encryption in use | Encryption in use refers to protecting data while it's being processed in memory. This policy includes encrypting data in runtime environments, using secure enclaves or memory isolation where appropriate. It ensures that sensitive data is not exposed during operations and is especially relevant for regulated or multi-tenant environments. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users or systems. This policy requires multi-factor authentication (MFA) for all user logins, including administrative access, to ensure that only authorized personnel gain access to secure systems. It applies to all login events, including UI and API authentication. |
| Authorization | Authorization determines what authenticated users are allowed to access. This policy enforces role-based access control (RBAC) to restrict users to only the resources and actions necessary for their role. It governs access to files, systems, and database operations, and helps minimize the impact of compromised accounts. |
| Accounting | Accounting refers to logging and monitoring user and system actions. This policy requires detailed audit trails for user logins, database changes, access to sensitive files, and permission modifications. These logs must be immutable, retained for at least one year, and reviewed regularly to detect and respond to suspicious behavior. |

**\***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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/25/2025 | Define coding standards with compliant and noncompliant code examples. | Edwin Jones |  |
| 1.2 | 06/15/2025 | Perform risk assessments, establish principles, tools, and automation processes, and define encryption and triple-a policies. | Edwin Jones |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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