**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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 12](#_Toc52464063)

[Coding Standard 6 14](#_Toc52464064)

[Coding Standard 7 16](#_Toc52464065)

[Coding Standard 8 18](#_Toc52464066)

[Coding Standard 9 20](#_Toc52464067)

[Coding Standard 10 22](#_Toc52464068)

[Defense-in-Depth Illustration 25](#_Toc52464069)

[Project One 25](#_Toc52464070)

[1. Revise the C/C++ Standards 25](#_Toc52464071)

[2. Risk Assessment 25](#_Toc52464072)

[3. Automated Detection 25](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 27](#_Toc52464076)

[7. Map the Principles 28](#_Toc52464077)

[Audit Controls and Management 29](#_Toc52464078)

[Enforcement 29](#_Toc52464079)

[Exceptions Process 29](#_Toc52464080)

[Distribution 30](#_Toc52464081)

[Policy Change Control 30](#_Toc52464082)

[Policy Version History 30](#_Toc52464083)

[Appendix A Lookups 30](#_Toc52464084)

[Approved C/C++ Language Acronyms 30](#_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 | Input validation ensures all data entering a system is checked for correctness, is complete, and is compliant with expected formats before processing. This prevents vulnerabilities by rejecting possible malicious or malformed inputs. |
| 1. Heed Compiler Warnings | Compiler warnings highlight potential issues in code, such as type mismatches or uninitialized variables that can lead to security vulnerabilities or runtime errors. By addressing these warnings developers can help prevent bugs, unintended behavior, and reduce security risk early on in the development process. |
| 1. Architect and Design for Security Policies | Security must be a priority from the beginning stages and integrated into the software architecture from the design phase, ensuring the system has good security principles such as authentication, least privilege, and a focus on defense in depth. This will ensure that security policies are embedded in the system’s structure and can reduce risk from the start. |
| 1. Keep It Simple | Simplicity not only in design but in implementation can reduce the areas of attack, make it easier to identify potential security issues, and makes it much easier to maintain, review, and test. |
| 1. Default Deny | By default systems should reject access to resources, functions, and data unless explicitly allowed. This approach can reduce the exposure to unauthorized access and reduce the potential for exploitation, and increase overall security. |
| 1. Adhere to the Principle of Least Privilege | All users and processes should only operate with minimum permissions to what is necessary to their tasks. By keeping permissions limited in structure can reduce the impact from compromised factors such as accounts or components. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data in transit can remove unwanted content and prevent malicious content, injection attacks, or cross-site scripting. This will help ensure safe data exchanged across systems. |
| 1. Practice Defense in Depth | Defense in Depth is applying multiple layers of security such as firewalls, encryption, and access controls to protect against threats. This style of layered approach improves overall defense and creates a failsafe if one of the layers is compromised. |
| 1. Use Effective Quality Assurance Techniques | Using techniques such as code review, testing, static analysis, and regular security audits can help detect vulnerabilities early in development. By following these practices can help ensure clean and secure code from the beginning. |
| 1. Adopt a Secure Coding Standard | By following secure coding guidelines can help avoid common vulnerabilities, such as buffer overflows. This ensures consistent and secure coding across all development and can reduce the risk of security flaws. |

### 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 signed integers do not result in overflow. (INT32-C)** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using fixed-width integer types prevents undefined behavior due to integer overflows or platform-dependent sizes. This ensures consistent, safe, arithmetic operations across platforms, reducing vulnerabilities like buffer overflows. |

| **Noncompliant Code** |
| --- |
| Using int for arithmetic can cause overflow on platforms with varying integer sizes, leading to unintended behavior. |
| int a = 2147483647; //max int on 32 bit system  int b = int a + 1; //overflow |

| **Compliant Code** |
| --- |
| Using int32\_t with explicit bounds checking prevents overflow. |
| #include <cstdint>  #include <limits>  Int32\_t a = 2147483647;  if (a < std::numeric\_limits<int32\_t>::max()){  int32\_t b = a + 1;  } |

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

| **Principles(s):**  **Principle 1 (Validate Input Data):** Ensuring integers do not overflow validates the integrity of arithmetic inputs, preventing malformed data from causing vulnerabilities.  **Principle 9 (Use Effective Quality Assurance Techniques):** Bounds checking and static analysis tools help detect potential overflows during development.  **Principle 10 (Adopt a Secure Coding Standard):** This standard directly aligns with secure coding practices to avoid common vulnerabilities like integer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-int32-c | Detects potential integer overflows in arithmetic operations. |
| Cppcheck | 2.12 | integerOverflow | Identifies risky integer operations that may lead to undefined behavior. |
| Coverity | 2023.6 | INTEGER\_OVERFLOW | Static analysis tool for detecting integer overflow vulnerabilities in C++ code. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Initialize All Variables (EXP53-CPP)** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Initializing variables to avoid undefined behavior from uninitialized data, which can lead to unpredictable results or security vulnerabilities. Initialization ensures consistent program behavior. |

| **Noncompliant Code** |
| --- |
| Uninitialized variables can contain arbitrary variables, leading to undefined behavior. |
| #include <iostream>  int main() {  int total;  std::cout << “Total: “ << total << ‘\n’; //Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| Initializing variables at declaration ensures predictable behavior and eliminates undefined states. |
| #include <iostream>  int main() {  int total = 0;  std::cout << “Total: “ << total << ‘\n’; //Safe output  return 0;  } |

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

| **Principles(s):**  **Principle 4 (Keep It Simple):** Initialize variables simplifies debugging and ensures predictable behavior.  **Principle 9 (Use Effective Quality Assurance Techniques):** Static analysis tools can detect uninitialized variables during code review.  **Principle 10 (Adopt a Secure Coding Standard):** This standard enforces consistent initialization to prevent vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-exp53-cpp | Flags uninitialized variables in C++ code. |
| Cppcheck | 2.12 | uninitvar | Detects uninitialized variables that may lead to undefined behavior. |
| Parasoft | 2024.2 | CERT\_CPP\_EXP53-a | Avoid use before initialization. |
| SonarQube | 9.9 | S2259 | Identifies uninitialized variables during static code analysis. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee Sufficient Space for Strings (STR31-C)** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Improper use of string manipulation functions can result in truncated output or buffer overflows, leading to data corruption, application crashes, or exploitable security vulnerabilities. Ensuring correct sizing and usage of string operations prevents these issues. |

| **Noncompliant Code** |
| --- |
| Code uses snprintf() but allocates insufficient space for the terminating null byte. |
| #include <cstdio>  int main() {  char buffer[5];  snprintf(buffer, sizeof(buffer), “Hello”); //”Hell”  printf(“%s\n”, buffer); //Undefined behavior  } |

| **Compliant Code** |
| --- |
| This version correctly sizes the buffer and checks for truncation |
| #include <cstdio>  #include <cstring>  int main() {  char buffer[6]; // Room for “Hello” + null terminator  int result = snprintf(buffer, sizeof(buffer), “Hello”);  if (result >= sizeof(buffer)) {  }  Printf(“%\n”, buffer); //Safe  } |

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

| **Principles(s):**  **Principle 1 (Validate Input Data):** Validating string sizes prevents overflows and ensures data integrity.  **Principle 8 (Practice Defense in Depth):** Proper string handling adds a layer of protection against buffer overflow attacks.  **Principle 10 (Adopt a Secure Coding Standard):** This standard aligns with secure string manipulation practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-str32-c | Detects unsafe string operations that may cause buffer overflows. |
| CodeSonar | 9.1 | LANG.MEM.BO | Buffer Overrun |
| Coverity | 2023.6 | BUFFER\_SIZE | Identifies potential buffer overflow vulnerabilities in string operation. |
| Cppcheck | 2.12 | bufferAccessOutOfBounds | Flags string operations exceeding allocated buffer sizes. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize Data Passed to Subsystems (STR02-C)** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Complex subsystems such as SQL engines can behave unpredictably when passed with untrusted input. Sanitizing and validating input before it reaches these subsystems helps prevent injection attacks, unauthorized access, and undefined behavior. |

| **Noncompliant Code** |
| --- |
| This code passes un-sanitized user input into an SQL string, which can lead to SQL Injection. |
| #include <iostream>  #include <string>  int main() {  std::string userInput = “’; DROP TABLE users; --“;  std::string query = “SELECT \* FROM users WHERE name = ‘” + userInput  + “’”;  std::cout << “Executing: “ << query << ‘\n’;  //Potential injection  } |

| **Compliant Code** |
| --- |
| This version uses a sanitization function to remove dangerous characters and enforces safe query construction |
| #include <iostream>  #include <string>  #include <regex>  std::string sanitizeInput(const std::string& input) {  return std::regex\_replace(input, std::regex(R”([‘”\\])”, “”);  }  int main() {  std::string userInput = sanitizeInput(“JohnDoe”);  std::string query = “SELECT \* FROM users WHERE name = ‘” + userInput  + “’”;  std::cout << “Executing: “ << query << ‘\n’;  } |

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

| **Principles(s):**  **Principle 1 (Validate Input Data):**  Sanitizing inputs ensures only valid data is processed by subsystems.  **Principle 7 (Sanitize Data Sent to Other Systems):** This standard directly implements data sanitization to prevent injection attacks.  **Principle 8 (Practice Defense in Depth):** Sanitization adds a protective layer against subsystem vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-str02-c | Flags unsanitized inputs passed to subsystems. |
| Coverity | 2023.6 | SQL\_INJECTION | Identifies SQL injection risks in code. |
| SonarQube | 9.9 | S2077 | Detects potential SQL injection vulnerabilities in string concatenation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and Handle Memory Allocation Errors (MEM52-CPP)** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Detecting and handling memory allocation failures can prevent undefined behavior and crashes. Checking for null pointers after allocation or using smart pointers like std::unique\_ptr ensures robust memory management. |

| **Noncompliant Code** |
| --- |
| This code attempts to allocate memory using new but does not check if the allocation succeeded. |
| #include <iostream>  int main() {  int\* data = new(std::nothrow) int[1000000000]; //Large allocation  \*data = 42; //Undefined behavior if data is nullptr  std::cout << \*data << std::endl;  delete[] data;  return 0;  } |

| **Compliant Code** |
| --- |
| This code checks whether the memory allocation is successful before using the pointer, preventing undefined behavior. |
| #include <iostream>  int main() {  int\* data = new(std::nothrow) int[1000000000]; //Large allocation  if (data == nullptr) {  std::cerr << “Memory allocation failed” << std::endl;  return 1;  }  \*data = 42;  std::cout << \*data << std::endl;  delete[] data;  return 0;  } |

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

| **Principles(s):**  **Principle 8 (Practice Defense in Depth):** Robust memory management adds a layer of protection against crashes and exploits.  **Principle 9 (Use Effective Quality Assurance Techniques):** Static analysis can detect allocation errors early.  **Principle 10 (Adopt a Secure Coding Standard):** This standard enforces safe memory handling practices. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-mem52-cpp | Detects unchecked memory allocations in C++ code. |
| Cppcheck | 2.12 | nullPointer | Identifies potential null pointer dereferences after allocation. |
| Coverity | 2023.6 | UNCHECKED\_ALLOC | Flags unhandled memory allocation failures. |
| Parasoft | 2024.2 | CERT\_CPP\_MEM52-a | Check return value of new. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Validate Object Lifetime (EXP54-CPP)** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Accessing objects outside their lifetime can lead to undefined behavior. Using assertions to validate that objects are within their valid lifetime before access ensures the program correctness and prevents security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Accessing a pointer without verifying its lifetime can cause undefined behavior. |
| #include <iostream>  void process(int \*ptr){  \*ptr = 10; //No check for object lifetime, potential invalid access  }  int main() {  int\*ptr = nullptr; //Pointer not initialized to valid object  process(ptr); //Undefined behavior  std::cout << “Value: ” << \*ptr << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Assertions validate that a pointer is non-null and within its lifetime. |
| #include <cassert>  #include <iostream>  void process(int \*ptr) {  assert(ptr != nullptr && “Pointer must be valid”);//Validate lifetime  \*ptr = 10;  }  int main() {  int value = 42;  int \*ptr = &value; //Pointer to validate object  process(ptr);  std::cout << “Value: “ << \*ptr << std::endl; //Safe access  return 0;  } |

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

| **Principles(s):**  **Principle 1 (Validate Input Data):** Assertions validate pointer validity, ensuring safe access.  **Principle 9 (Use Effective Quality Assurance Techniques):** Assertions enable early detection of lifetime issues.  **Principle 10 (Adopt a Secure Coding Standard):** This standard enforces lifetime validation practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-exp54-cpp | Detects potential invalid object accesses. |
| Cppcheck | 2.12 | invalidPointer | Flags dereferences of pointers with invalid lifetimes. |
| Coverity | 2023.6 | INVALID\_MEMORY\_ACCESS | Identifies accesses to objects outside their lifetimes. |
| Parasoft | 2024.2 | CERT\_CPP-EXP54-a | Do not use resources that have been freed. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle All Exceptions (ERR51-CPP)** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Catching all possible exceptions to prevent program termination or undefined behavior. Properly handling exceptions ensures resources are released and the program remains stable, reducing vulnerabilities such as resource leaks or crashes. |

| **Noncompliant Code** |
| --- |
| Failing to handle exceptions can lead to program crashes or resource leaks. |
| #include <iostream>  #include <stdexcept>  Void risky\_operation(int size){  int \*arr = new int[size]; //May throw std::bad\_alloc  if (size < 0){  throw std::invalid\_argument(“Negative size”);  }  arr[0] = 42; //Unhandled exception  }  int main() {  risky\_operation(-1); //Uncaught exception causes termination  std::cout <<”Operation completed” << std::endl; //Unreachable  return 0;  } |

| **Compliant Code** |
| --- |
| Catching all exceptions ensures proper resource cleanup and stable program execution. |
| #include <iostream>  #include <stdexcept>  #include <memory>  void risky\_operation(int size) {  std::unique\_ptr<int[]> arr = std::make\_unique<int[]>(size);  //Auto Cleaned  if (size < 0) {  throw std::invalid\_argument(“Negative size”);  }  arr[0] = 42;  }  int main() {  try {  risky\_operation(-1); // Exception caught safely  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid argument: " << e.what() << std::endl;  } catch (const std::bad\_alloc& e) {  std::cerr << "Memory allocation failed: " << e.what() <<  std::endl;  } catch (...) {  std::cerr << "Unknown exception caught" << std::endl;  }  std::cout << "Operation handled safely" << std::endl;  return 0;  } |

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

| **Principles(s):**  **Principle 8 (Practice Defense in Depth):** Exception handling adds resilience against unexpected failures.  **Principle 9 (Use Effective Quality Assurance Techniques):** Testing and code reviews ensure proper exception handling.  **Principle 10 (Adopt a Secure Coding Standard):** This standard enforces robust error management practices. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probably | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-err51-cpp | Detects unhandled exceptions in C++ code. |
| Coverity | 2023.6 | UNCAUGHT\_EXCEPTION | Identifies code paths with unhandled exceptions. |
| Parasoft | 2024.2 | CERT\_CPP-ERR51-a | Always catch exceptions. |
| SonarQube | 9.9 | S2170 | Flags missing exception handling in critical code paths. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Avoid Undefined Evaluation Order (EXP50-CPP)** |
| --- | --- | --- |
| Data Validation | STD-008-CPP | Do not rely on the order of evaluation for function arguments or subexpressions, as it is undefined in C++ and can lead to unintended results. For data validation, this ensures that validation logic produces consistent outcomes, preventing errors or vulnerabilities due to incorrect assumptions about evaluation order. |

| **Noncompliant Code** |
| --- |
| Relying on the order of evaluation in a function call can lead to undefined behavior, causing inconsistent validation results. |
| #include <iostream>  int validate\_input(int value) {  std::cout << "Validating: " << value << std::endl;  return value > 0 ? value : 0;  }int process(int x, int y) {  return x + y;  }int main() {  int i = 0;  // Undefined order of evaluation for validate\_input calls  int result = process(validate\_input(i++), validate\_input(i++));  std::cout << "Result: " << result << std::endl; // Unpredictable outcome  return 0;  } |

| **Compliant Code** |
| --- |
| Evaluating expressions in a defined order ensures predictable data validation. |
| #include <iostream>  int validate\_input(int value) {  std::cout << "Validating: " << value << std::endl;  return value > 0 ? value : 0;  }  int process(int x, int y) {  return x + y;  }  int main() {  int i = 0;  // Evaluate each expression explicitly to avoid undefined order  int x = validate\_input(i);  i++;  int y = validate\_input(i);  int result = process(x, y);  std::cout << "Result: " << result << std::endl;  // Predictable outcome  return 0;  } |

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

| **Principles(s):**  **Principle 1 (Validation Input Data):** Defined evaluation order ensures consistent validation results.  **Principle 4 (Keep It Simple):** Explicit evaluation simplifies code and reduces errors.  **Principle 10 (Adopt a Secure Coding Standard):** This standard enforces predictable behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Low | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-exp50-cpp | Detects undefined evaluation order in expressions. |
| Cppcheck | 2.12 | undefinedBehavior | Flags expressions with undefined evaluation order. |
| Coverity | 2023.6 | UNDEFINED\_EVAL\_ORDER | Identifies code with unpredictable evaluation sequences. |
| Parasoft | 2024.2 | CERT\_CPP-EXP50-a | The value of an expression shall be under the same under any order of evaluation that the standard permits. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close Files When No Longer Needed (FIO51-CPP)** |
| --- | --- | --- |
| File Handling | STD-009-CPP | Closing files when they are no longer needed to prevent resource leaks and ensure system stability. Failing to close files can exhaust file descriptors, leading to denial-of-service vulnerabilities or data corruption. |

| **Noncompliant Code** |
| --- |
| Failing to close a file after use can leak resources, potentially exhausting file descriptors and causing system failures. |
| #include <cstdio>  #include <iostream>  void write\_data(const char\* filename) {  FILE\* file = std::fopen(filename, "w"); // Open file  if (file) {  std::fputs("Example data", file); // Write data  // File not closed, resource leak  }  }int main() {  write\_data("output.txt");  std::cout << "File written" << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using ‘std::fstream’ with RAII ensures files are automatically closed when out of scope, preventing resource leaks. |
| #include <fstream>  #include <iostream>  void write\_data(const std::string& filename) {  std::ofstream file(filename); // RAII manages file  if (file.is\_open()) {  file << "Example data"; // Write data  // File auto-closed when file goes out of scope  }  }  int main() {  write\_data("output.txt");  std::cout << "File written" << std::endl;  return 0;  } |

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

| **Principles(s):**  **Principle 6 (Adhere to Principle of Least Privilege):** Closing files limits resource usage to what is necessary.  **Principle 8 (Practice Defense in Depth):** RAII file handling adds a layer of protection against resource leaks.  **Principle 10 (Adopt a Secure Coding Standard):**  This standard enforces proper resource management. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-fio51-cpp | Detects unclosed file handles in C++ code. |
| Cppcheck | 2.12 | resourceLeak | Flags unclosed file descriptors that may cause leaks. |
| Coverity | 2023.6 | RESOURCE\_LEAK | Identifies unclosed files leading to resource exhaustion. |
| Parasoft | 2024.2 | CERT\_CPP-FIO51-a | Ensure resources are freed. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Ensure Locks are Released on Exceptions (CON51-CPP)** |
| --- | --- | --- |
| Concurrency | STD-010-CPP | Locks in multithreaded programs need released even when exceptions occur to prevent deadlocks. Using RAII-based lock guards ensures automatic lock release, maintaining thread safety and preventing resource contention. |

| **Noncompliant Code** |
| --- |
| Manually locking and unlocking a mutex without handling exceptions can leave the mutex locked, causing deadlocks in concurrent programs. |
| #include <mutex>  #include <stdexcept>  #include <iostream>  std::mutex mtx;  int shared\_data = 0;void update\_data(int value) {  mtx.lock(); // Manual lock  if (value < 0) {  throw std::invalid\_argument("Negative value"); // Exception leaves lock held  }  shared\_data = value;  mtx.unlock(); // Unreachable if exception occurs  }int main() {  try {  update\_data(-1); // Causes exception, mutex remains locked  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  // Other threads may deadlock trying to acquire mtx  std::cout << "Shared data: " << shared\_data << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using ‘std::lock\_guard’ ensures the mutex is automatically unlocked when the scope is exited, even on exceptions, preventing deadlocks. |
| #include <mutex>  #include <stdexcept>  #include <iostream  std::mutex mtx;  int shared\_data = 0;  void update\_data(int value) {  std::lock\_guard<std::mutex> lock(mtx); // RAII lock, auto-released  if (value < 0) {  throw std::invalid\_argument("Negative value"); // Lock released on exception  }  shared\_data = value;  }  int main() {  try {  update\_data(-1); // Exception handled, lock auto-released  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  std::cout << "Shared data: " << shared\_data << std::endl; // Safe access  return 0;  } |

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

| **Principles(s):**  **Principle 6 (Adhere to the Principle of Least Privilege):** RAII locks limit resources access to necessary scopes.  **Principle 8 (Practice Defense in Depth):**  Automatic lock release adds reliability in concurrent systems.  **Principle 10 (Adopt a Secure Coding Standard):**  This standard enforces safe concurrency practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Cert-con51-cpp | Detects unhandled locks in exception paths. |
| Cppcheck | 2.12 | lockNoUnlock | Flags locks not released in exception scenarios. |
| Coverity | 2023.6 | LOCK\_NOT\_RELEASED | Identifies mutexes left locked due to exceptions. |
| Parasoft | 2024.2 | CERT\_CPP-CON51-a | Do not call lock() directly on a mutex. |

### 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.

**Assessing and Planning:** Use static analysis tools during code review to identify violations of coding standards. Tools like Clang-Tidy and Cppcheck could be integrated with into CI/CD pipelines to flag issues before code is finalized.

**Designing and Building:** Incorporate automated testing frameworks to perform dynamic and static analysis during the build phase. Tools should check for SQL injection risks, unhandled exceptions, and resource leaks. Tools like SonarQube can check for these risks and ensure compliance with standards like STD-004-CPP and STD-007-CPP.

**Maintaining System Integrity:**  Deploy continuous monitoring tools to audit runtime behavior, such as file descriptor leaks or concurrency issues. Tools like Coverity can generate reports for the risk management committee to review compliance regularly.

By following these outlines we can ensure compliance with Green Pace Secure Development Policy, automation must be integrated into the existing DevOps pipeline which can create a robust process. By continuing to access, plan, design, build, and maintain will create better and higher system integrity. Green Pace can enforce standards automatically during Software Development Life Cycles reducing oversights and keeping compliant with security policy.

### 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 | Likely | Medium | P18 | L1 |
| STD-002-CPP | High | Probable | Low | P12 | L1 |
| STD-003-CPP | High | Likely | Medium | P9 | L1 |
| STD-004-CPP | High | Likely | Medium | P9 | L2 |
| STD-005-CPP | High | Likely | Low | P27 | L1 |
| STD-006-CPP | High | Probable | Low | P6 | L2 |
| STD-007-CPP | Low | Probable | Medium | P6 | L2 |
| STD-008-CPP | Medium | Probable | Low | P8 | L2 |
| STD-009-CPP | Medium | Unlikely | Low | P2 | L3 |
| STD-010-CPP | Low | 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 disk or in databases, preventing unauthorized access if storage is compromised. Using encryption as AES-256 can encrypt sensitive data stored in databases or files. This will ensure all persistent data can comply with regulations and prevent data breaches. |
| Encryption in flight | Encryption in flight secures data being transmitted over networks, this can prevent interception or tampering. Use encryption in flight for all network communications including web traffic, and enforcing HTTPS for all web interfaces and validation certificates. This policy applies to all data being transmitted between clients, servers, and third-party services to protect against man-in-the-middle attacks. This policy applies to all data transmitted between clients, servers, and third-party services to protect against these man-in-the-middle attacks. |
| Encryption in use | Encryption in use protects data while it is being processed in memory. Use secure encryption for sensitive computations and ensure memory is cleared after use which can prevent data leaks. This policy applies to applications processing sensitive data which can protect against memory-based attacks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies user or system identity before granting access. By implementing multi-factor authentication for all user logins and having secure login information (passwords). This policy should apply to all user and system access points, including logins and administrative interfaces, which will prevent unauthorized access. |
| Authorization | Authorization is used to define user or system permissions based on their roles. By placing role based access to manage the levels of access each user can access. This can also give permissions and restrict access to sensitive information and can limit the access to database modifications. |
| Accounting | Accounting tracks and logs all user and system activities. By logging all users login and activities which will allow you to ensure that all users stay within their bounds and logs can be checked for any issues and keep compliant for regular maintenance. This will allow compliance and maintain detection for suspension activities. |

**\***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 |  |
| 2.0 | 07/20/2025 | Completed Module 3 Milestone One | Zachariah Garrison | [Insert text.] |
| 3.0 | 8/10/2025 | Completed Project One | Zachariah Garrison | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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