**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 | Validating input data ensures that only correctly formatted data enters the system, which prevents malicious inputs from causing harm. This step is crucial in preventing attacks like SQL injection, buffer overflows, and other injection attacks by ensuring that all user inputs conform to expected formats before being processed. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential vulnerabilities or coding issues that could be exploited. By paying attention to and resolving these warnings, developers can proactively address security flaws before they become significant problems, enhancing the overall robustness of the application. |
| 1. Architect and Design for Security Policies | Security should be integrated into the architecture and design of systems from the outset. This means incorporating security policies that define how data is protected, how access is controlled, and how risks are mitigated, ensuring that security is a fundamental part of the system's structure. |
| 1. Keep It Simple | Complexity often leads to errors, which can introduce vulnerabilities. By keeping the design and implementation simple, it's easier to understand, test, and secure the system. Simple code is less likely to contain hidden bugs and is easier to maintain and audit. |
| 1. Default Deny | The principle of "default deny" means that access is denied unless explicitly allowed. This approach minimizes the attack surface by ensuring that no unnecessary permissions are granted, and only the required permissions are given to users or processes. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege restricts users and processes to the minimal level of access necessary to perform their functions. This reduces the risk of unauthorized access or damage if an account or process is compromised, limiting the potential impact of security breaches. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems ensures that it is free from malicious content. This prevents the introduction of vulnerabilities when interacting with external systems or components, such as databases or third-party services, protecting against attacks like SQL injection or cross-site scripting (XSS). |
| 1. Practice Defense in Depth | Defense in depth involves layering multiple security measures to protect against various threats. By employing multiple layers of defense, such as firewalls, encryption, authentication, and intrusion detection systems, an attacker must bypass multiple barriers to compromise a system, enhancing overall security. |
| 1. Use Effective Quality Assurance Techniques | Implementing thorough quality assurance (QA) techniques, including code reviews, automated testing, and security assessments, helps identify and eliminate vulnerabilities early in the development process. Effective QA ensures that the code meets security standards and performs as expected under various conditions. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard provides guidelines for writing code that is free from common vulnerabilities. It establishes best practices that developers can follow to ensure that security is consistently addressed throughout the development lifecycle, reducing the likelihood of introducing 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** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | CTR-050-CPP | Guarantee that container indices and iterators are within the valid range. |

| **Noncompliant Code** |
| --- |
| This example has an infinite loop with a logical error that causes out-of-bounds access, or iterator overflow: |
| void print( const vector<int>& vec){  for(int i = 0; ++i){  if(vec[i]= 23){  break;  }else{  cout << “Not “ << I << endl;  }  }  } |

| **Compliant Code** |
| --- |
| Here, the loop correctly manages the index and checks before accessing the vector: |
| void print(const std::vector<int>& vec) {  for (int i = 0; i < vec.size(); ++i) {  if (vec[i] == 23) {  break;  } else {  std::cout << "Not " << i << std::endl;  }  }  } |

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

| **Principles(s):**  **Validate Input Data**: Supports data integrity by confirming that indices and iterators are within valid boundaries before use.  **Adopt a Secure Coding Standard**: Ensures indices and iterators remain within valid ranges to prevent undefined behavior, reducing vulnerability risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S7034 <https://rules.sonarsource.com/cpp/RSPEC-6171/?search=Container%20Bounds> | This rule checks for out-of-bound access in arrays and containers, ensuring that indices remain valid. |
| Valgrind | 3.17.0 | Memcheck <https://valgrind.org/docs/manual/mc-manual.html> | Valgrind’s Memcheck detects invalid memory access, including out-of-bounds reads/writes that could occur due to incorrect iterator or index values, complementing static bounds checks. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT-050-CPP | Ensure that data values are within expected ranges. |

| **Noncompliant Code** |
| --- |
| In this example, an integer multiplication can cause an overflow: |
| void multiply(int a, int b) {  int result = a \* b;  // Potential overflow if a and b are large enough  } |

| **Compliant Code** |
| --- |
| The compliant code checks for potential overflow before performing the multiplication: |
| void multiply(int a, int b) {  if (a > 0 && b > 0 && a > (INT\_MAX / b)) {  // Handle the overflow case  } else {  int result = a \* b;  }  } |

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

| **Principles(s):**  **Adopt a Secure Coding Standard**: This standard mandates checks for overflow conditions, ensuring that integer operations do not result in undefined behavior or security vulnerabilities.  **Validate Input Data**: Ensuring that data values are within expected ranges is a fundamental aspect of input validation, preventing erroneous or malicious data from causing overflow conditions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S5945  <https://rules.sonarsource.com/cpp/RSPEC-6191/?search=data%20overflow> | This rule checks for arithmetic operations that could result in overflows or underflows, ensuring that data values remain within valid ranges. |
| Fortify Static Code Analyzer | 20.2.0 | Dataflow Analysis | Fortify’s Dataflow Analysis checks for vulnerabilities in data validation and arithmetic operations, preventing integer overflows and underflows during runtime by tracing how data moves through the application. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR-051-CPP | Do not attempt to create a std::string from a null pointer. |

| **Noncompliant Code** |
| --- |
| In this example, a function takes a const char\* pointer and attempts to create a std::string without checking if the pointer is null: |
| void processInput(const char\* input) {  std::string str(input); // Undefined behavior if input is nullptr  } |

| **Compliant Code** |
| --- |
| Here, the function checks if the pointer is null before attempting to create a std::string: |
| void processInput(const char\* input) {  std::string str;  if (input != nullptr) {  str = std::string(input); // Safe: only construct std::string if input is not null  } else {  // Handle the null case  }  } |

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

| **Principles(s):**  **Heed Compiler Warnings**: Compiler warnings can often detect the risk of creating a std::string from a null pointer. Addressing these warnings proactively helps prevent undefined behavior and ensures safer code.  **Keep It Simple**: By simply checking if a pointer is null before using it, the code remains straightforward and less prone to errors, contributing to overall system stability and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S2637  <https://rules.sonarsource.com/cpp/RSPEC-2259/> | Detects potential null pointer dereferences, ensuring that std::string is not constructed from a null pointer. |
| Clang Static Analyzer | 12.0.0 | NullDereference  <https://clang.llvm.org/docs/analyzer/checkers.html#core-nulldereference-c-c-objc> | Clang Static Analyzer detects null pointer dereferences before runtime, ensuring that a std::string is not constructed from a null pointer, preventing undefined behavior in production. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS-007-CPP | Sanitize all user inputs passed to SQL queries. |

| **Noncompliant Code** |
| --- |
| The code directly incorporates unsanitized user input into an SQL query, making it vulnerable to SQL injection. |
| void executeQuery(const char\* userInput) {  char query[256];  snprintf(query, sizeof(query), "SELECT \* FROM users WHERE name = '%s';", userInput);  db\_execute(query);  } |

| **Compliant Code** |
| --- |
| The compliant code uses parameterized queries, which safely binds user inputs, preventing SQL injection attacks. |
| void executeQuery(const char\* userInput) {  const char\* query = "SELECT \* FROM users WHERE name = ?";  db\_prepare(query);  db\_bind(1, userInput);  db\_execute();  } |

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

| **Principles(s):**  **Sanitize Data Sent to Other Systems**: This principle directly applies, ensuring that any data sent to an SQL database is sanitized to prevent malicious inputs from executing unintended commands.  **Practice Defense in Depth**: By using parameterized queries, this standard adds an additional layer of security, protecting the system even if other defenses, such as input validation, fail. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S2077 | Detects cases where user inputs are not properly sanitized before being passed to SQL queries, preventing SQL injection vulnerabilities. |
| OWASP ZAP | 2.10.0 | SQL Injection Scanner  <https://www.zaproxy.org/docs/alerts/2/> | OWASP ZAP performs dynamic analysis on web applications, actively scanning for unsanitized user inputs in SQL queries, helping to detect SQL injection vulnerabilities that static analysis might miss. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-050-CPP | Ensure that dynamically allocated memory is properly managed. |

| **Noncompliant Code** |
| --- |
| In this example, memory is allocated, freed, and then accessed again, leading to undefined behavior: |
| void process() {  int\* data = new int[100];  delete[] data;  data[0] = 42; // Undefined behavior  } |

| **Compliant Code** |
| --- |
| In the compliant code, memory is properly managed by ensuring it is not accessed after being freed: |
| void process() {  int\* data = new int[100];  data[0] = 42;  delete[] data;  } |

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

| **Principles(s):**  **Use Effective Quality Assurance Techniques**: Ensuring proper memory management can be effectively achieved through rigorous testing and code reviews, which help detect issues like use-after-free errors.  **Keep It Simple:** Simplifying memory management by ensuring that memory is not accessed after being freed reduces the risk of undefined behavior, making the code easier to maintain and less error-prone. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S5025  <https://rules.sonarsource.com/cpp/RSPEC-3584/> | Identifies cases where dynamically allocated memory is not properly deallocated, preventing memory leaks. |
| AndressSanitizer | 11.1.0 | Memory Leak Detection  <https://clang.llvm.org/docs/AddressSanitizer.html> | AddressSanitizer dynamically detects memory leaks and improper memory management, providing detailed runtime error reports when dynamically allocated memory is not properly deallocated, complementing static checks for memory leaks. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MSC-011-CLG | Incorporate diagnostic tests using assertions. |

| **Noncompliant Code** |
| --- |
| The code uses an assertion to check for a null pointer, which could occur due to external factors. |
| void process(int\* ptr) {  assert(ptr != nullptr); // Misuse of assertion for runtime error  \*ptr = 42;  } |

| **Compliant Code** |
| --- |
| The code correctly handles potential runtime errors using proper error handling, ensuring that the program behaves correctly in all situations. |
| void process(int\* ptr) {  if (ptr == nullptr) {  throw std::invalid\_argument("Null pointer passed to process");  }  \*ptr = 42;  } |

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

| **Principles(s):**  **Architect and Design for Security Policies:** This principle emphasizes that security measures, such as proper error handling instead of relying solely on assertions, should be integrated into the design to handle unexpected situations securely.  **Default Deny:** By handling null pointers with explicit exceptions rather than assertions, the code adheres to a "default deny" approach, ensuring that potentially harmful operations are not permitted unless explicitly safe. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S7012  <https://rules.sonarsource.com/cpp/RSPEC-7012/?search=assertions> | This rule ensures that assertions are only used for debugging and diagnostic purposes, not for runtime error handling. |
| CPPcheck | 2.4.1 | Misuse of Assertions  <http://cppcheck.sourceforge.net/> | Cppcheck’s rule detects improper use of assertions in production code, ensuring that assertions are only used for debugging purposes and are not relied upon for runtime error handling. |

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR-051-CPP | Handle all exceptions. |

| **Noncompliant Code** |
| --- |
| In this example, an exception is thrown but not caught, leading to program termination: |
| void process() {  throw std::runtime\_error("Error occurred");  // No try-catch block to handle the exception  } |

| **Compliant Code** |
| --- |
| Here, the exception is properly caught and handled, ensuring the program can continue or fail gracefully: |
| void process() {  try {  throw std::runtime\_error("Error occurred");  } catch (const std::exception& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  // Handle the exception appropriately  }  } |

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

| **Principles(s):**  **Practice Defense in Depth:** Handling exceptions ensures that the system has multiple layers of defense, allowing for graceful failure and error recovery rather than allowing a single exception to cause a full program crash.  **Use Effective Quality Assurance Techniques:** Proper exception handling allows for comprehensive testing and debugging, ensuring the program behaves correctly even when unexpected errors occur. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | Medium | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S1035  <https://rules.sonarsource.com/cpp/RSPEC-1044/> | Detects unhandled exceptions, ensuring that all exceptions are caught and handled appropriately. |
| Fortify Static Code Analyzer | 20.2.0 | Control Flow Analysis | Fortify’s Control Flow Analysis tool checks for unhandled exceptions in complex code paths, ensuring that all thrown exceptions are properly caught and handled, reducing the risk of unexpected program termination. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory** | MEM-051-CPP | Properly deallocate dynamically allocated memory when no longer needed. |

| **Noncompliant Code** |
| --- |
| The code allocates memory but fails to deallocate it, leading to a memory leak. |
| void process() {  int\* data = new int[10];  // Further processing  } |

| **Compliant Code** |
| --- |
| The compliant code ensures that memory is deallocated properly after use, preventing memory leaks. |
| void process() {  int\* data = new int[10];  // Further processing  delete[] data; // Correctly deallocates the memory  } |

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

| **Principles(s):**  **Keep It Simple**: Ensuring that memory is properly deallocated when no longer needed simplifies the code, making it easier to understand and maintain, which reduces the likelihood of memory leaks.  **Adopt a Secure Coding Standard**: Proper memory management is a core aspect of secure coding, preventing issues like memory leaks that can lead to degraded performance or even system crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S5025  <https://rules.sonarsource.com/cpp/RSPEC-7012> | Ensures that all dynamically allocated resources are properly released after use, preventing memory leaks. |
| Valgrind | 3.17.0 | MemCheck  <https://valgrind.org/docs/manual/mc-manual.html> | Valgrind’s Memcheck tool ensures that dynamically allocated memory is properly freed at runtime, preventing memory leaks by reporting errors when memory is not properly deallocated. |

#### 

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Declaration** | DCL-060-CPP | Avoid object definitions in headers. |

| **Noncompliant Code** |
| --- |
| Defining static objects in a header file leads to multiple definitions. |
| // Header file: my\_header.h  static int counter = 0; // Defined will be defined again in cpp |

| **Compliant Code** |
| --- |
| The object is declared in the header and defined in a single source file, ensuring proper linkage. |
| // Header file: my\_header.h  extern int counter; //declaration only  // Source file: my\_source.cpp  int counter = 0; // Single definition in source file |

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

| **Principles(s):**  **Architect and Design for Security Policies**: Properly separating declarations and definitions in headers and source files is essential for ensuring that the code is modular, maintainable, and free from linkage issues that can cause unexpected behavior.  **Heed Compiler Warnings**: Compilers often warn about multiple definitions caused by improper declarations in headers. Addressing these warnings helps prevent potential runtime errors and undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | <https://rules.sonarsource.com/cpp/RSPEC-978/?search=define%20header> | Detects cases where object definitions are included in headers, preventing multiple definitions errors. |
| Clang-Tidy | 12.0.0 | Header Issue  <https://clang.llvm.org/extra/clang-tidy/> | Clang-Tidy checks for object definitions in header files, ensuring that only declarations appear in headers, preventing multiple definitions that lead to linkage errors. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Miscellaneous** | MSC-053-CPP | Do not return from a function declared [[noreturn]]. |

| **Noncompliant Code** |
| --- |
| The function is marked [[noreturn]] but may return if terminate is false, leading to undefined behavior. |
| [[noreturn]] void terminateOrContinue(bool terminate) {  if (terminate) {  std::exit(0);  }  // Undefined behavior: function might return if `terminate` is false  } |

| **Compliant Code** |
| --- |
| The compliant function ensures that it either exits the program or throws an exception, thus never returning to the caller. |
| [[noreturn]] void terminateOrContinue(bool terminate) {  if (terminate) {  std::exit(0);  } else {  throw std::runtime\_error("Continuing execution is not allowed");  }  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Rule:S5267  <https://rules.sonarsource.com/cpp/RSPEC-5267> | Ensures that functions declared with [[noreturn]] do not accidentally return, preventing undefined behavior. |
| CppCheck | 2.4.1 | NonReturning  <http://cppcheck.sourceforge.net/> | Cppcheck’s NonReturning rule ensures that functions marked [[noreturn]] behave as intended and do not accidentally return, preventing undefined behavior when these functions are used. |

### 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 plays a critical role in ensuring compliance with Green Pace's security standards throughout the software development lifecycle. Given the company's established DevOps infrastructure, security automation should be integrated into every phase of the DevSecOps pipeline.

In the planning and design phases, tools such as static analysis plugins for Integrated Development Environments (IDEs) can automatically detect potential vulnerabilities in real-time, allowing developers to address issues early. For instance, IDEs like Visual Studio or Eclipse can integrate with SonarQube or Clang-Tidy to highlight problems such as invalid memory access or improper input validation during coding. This helps ensure compliance from the outset.

As the project progresses into build and pre-production, Continuous Integration (CI) systems should run automated tests to verify the security and integrity of the codebase. Automated testing tools like Valgrind and AddressSanitizer can be used to check for memory leaks, buffer overflows, and other common C++ vulnerabilities. During this phase, ensuring that all critical components are secure before moving to production is key, with security scans being a required part of every build cycle.

Once in the production phase, automation tools can continue to monitor the application. Automated security scans, like those provided by OWASP ZAP, can perform dynamic security testing (DAST), identifying potential runtime vulnerabilities, such as SQL injection. Additionally, tools like SonarQube can continuously monitor code quality and security, alerting the team to new risks as the code evolves.

By integrating automation throughout the entire pipeline—from code writing to deployment—Green Pace can ensure that security remains a priority at every step, reducing manual effort while maintaining a robust 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 |
| --- | --- | --- | --- | --- | --- |
| CTR-050-CPP | High | Likely | High | High | 1 |
| DCL-060-CPP | High | Unlikely | High | Medium | 2 |
| ERR-051-CPP | Low | Probable | Medium | Medium | 1 |
| IDS-007-CPP | High | Probable | Medium | High | 2 |
| INT-050-CPP | High | Probable | Medium | High | 1 |
| MEM-050-CPP | High | Likely | Medium | High | 1 |
| MEM-051-CPP | High | Likely | Medium | High | 1 |
| MSC-011-CLG | Low | Unlikely | High | High | 1 |
| MSC-053-CPP | Medium | Unlikely | Low | High | 1 |
| STR-051-CPP | High | Likely | Medium | Medium | 1 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest secures data that is stored on physical media, such as hard drives, databases, and other storage systems, by encrypting it when it is not actively being accessed. This measure protects sensitive data from unauthorized access if storage devices are lost, stolen, or compromised. Implementing strong encryption algorithms, like AES-256, ensures that even if the storage medium is breached, the data remains unreadable to unauthorized individuals. |
| Encryption in flight | Encryption in flight secures data as it travels across networks, preventing interception by unauthorized parties. This is critical for maintaining confidentiality and integrity during data transmission, especially when dealing with sensitive information over the internet or internal networks. Protocols like TLS are employed to encrypt data in transit, making this policy essential for any communication between clients, servers, and APIs. |
| Encryption in use | Encryption in use ensures that data remains protected while being processed, a crucial measure for safeguarding sensitive operations that involve manipulating confidential data. This is particularly relevant in scenarios where data could be vulnerable to memory scraping or similar attacks. Techniques such as homomorphic encryption or secure enclaves are used to maintain encryption even during data processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users, devices, or systems, ensuring that only legitimate entities can access resources. This first step in the Triple-A framework is vital for securing access to systems and data, using methods like passwords, multi-factor authentication, and biometrics to confirm identities. Authentication is the gatekeeper that prevents unauthorized access, forming the foundation of secure access control. |
| Authorization | Authorization determines what authenticated users can do within the system, enforcing policies that dictate access levels and permissions. It ensures that users can only perform actions they are explicitly allowed to, preventing unauthorized activities. This process includes role-based access controls and other methods to restrict access based on user roles and contextual factors like location and time. |
| Accounting | Accounting tracks user activities within the system, logging access and actions to provide an audit trail. This component is crucial for maintaining transparency and accountability, enabling administrators to monitor access patterns, detect suspicious activities, and ensure compliance with security policies. Accounting is indispensable for auditing, incident response, and ongoing security management. |

### Map the Principles

**CTR-050-CPP: Guarantee container indices and iterators are within valid ranges.**

* **Validate Input Data (1)**: Ensuring indices and iterators are within valid ranges prevents errors from invalid or malicious data, aligning with the principle of validating all input data to protect against unexpected behavior.
* **Adopt a Secure Coding Standard (10)**: This standard enforces best practices to prevent vulnerabilities like buffer overflows, which are common issues when indices are out of range.

**INT-050-CPP: Ensure data values are within expected ranges.**

* **Validate Input Data (1)**: Checking that data values are within expected ranges is a direct application of input validation, preventing overflow errors that could lead to security vulnerabilities.
* **Heed Compiler Warnings (2)**: Compilers often provide warnings about potential overflows, so adhering to these warnings supports maintaining data integrity.

**STR-051-CPP: Do not create a std::string from a null pointer.**

* **Heed Compiler Warnings (2)**: Creating a std::string from a null pointer can result in undefined behavior, which compilers may warn against. Addressing these warnings is crucial for preventing runtime errors.
* **Keep It Simple (4)**: By checking if a pointer is null before using it, the code is simplified, making it less prone to errors and easier to maintain.

**IDS-007-CPP: Sanitize all user inputs in SQL queries.**

* **Sanitize Data Sent to Other Systems (7)**: This principle directly applies to preventing SQL injection attacks by ensuring that all user inputs are properly sanitized before being used in SQL queries.
* **Practice Defense in Depth (8)**: Using parameterized queries adds an extra layer of security, ensuring that the system remains protected even if other defenses fail.

**MEM-050-CPP: Properly manage dynamically allocated memory.**

* **Use Effective Quality Assurance Techniques (9)**: Effective quality assurance practices, such as testing and code reviews, help identify and correct improper memory management, preventing potential security vulnerabilities.
* **Keep It Simple (4)**: Simplifying memory management practices reduces the likelihood of errors like use-after-free, making the code easier to maintain and more secure.

**MSC-011-CLG: Use assertions for diagnostic tests.**

* **Architect and Design for Security Policies (3)**: Integrating assertions into the design ensures that unexpected conditions are caught early and handled securely, which is key for robust security policies.
* **Default Deny (5)**: Assertions help enforce the default deny principle by catching invalid states early, preventing the execution of potentially harmful code.

**ERR-051-CPP: Handle all exceptions.**

* **Practice Defense in Depth (8)**: Proper exception handling adds multiple layers of defense, allowing the system to fail gracefully and recover securely.
* **Use Effective Quality Assurance Techniques (9)**: Handling exceptions comprehensively ensures that the system is robust and behaves correctly even when unexpected errors occur.
* **Sanitize Data Sent to Other Systems (6)**: By handling exceptions properly, the code can ensure that any data being processed or transmitted during an exception is properly sanitized to prevent further issues.

**MEM-051-CPP: Deallocate dynamically allocated memory when no longer needed.**

* **Keep It Simple (4)**: Ensuring that memory is properly deallocated simplifies the code, making it easier to manage and reducing the risk of memory leaks.
* **Adopt a Secure Coding Standard (10)**: Proper memory management is a key aspect of secure coding, preventing issues like memory leaks that can lead to degraded performance or security vulnerabilities.

**DCL-060-CPP: Avoid object definitions in headers.**

* **Architect and Design for Security Policies (3)**: Properly separating declarations and definitions in headers and source files ensures that the code is modular, maintainable, and free from linkage issues that can cause undefined behavior.
* **Heed Compiler Warnings (2)**: Compilers often warn about multiple definitions caused by improper declarations in headers, supporting the need for this standard to maintain proper code linkage.

**MSC-053-CPP: Do not return from a function declared [[noreturn]].**

* **Use Effective Quality Assurance Techniques (9)**: Ensuring that functions marked [[noreturn]] do not return prevents undefined behavior and maintains the integrity of program execution.
* **Adopt a Secure Coding Standard (10)**: Following this standard helps prevent logical errors, ensuring that the function's behavior is consistent with its declaration.

**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 | 08/27/2024 | Risk Assessment, Standards, and Policies | Dennis Ward |  |
| 1.2 | 10/09/2024 | Review | Dennis Ward |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

References

SonarSource. (n.d.). *Rules explorer for C++*. SonarSource. Retrieved October 10, 2024, from <https://rules.sonarsource.com/cpp>

Valgrind Developers. (n.d.). *Valgrind: Memcheck documentation*. Valgrind.org. Retrieved October 10, 2024, from <https://valgrind.org/docs/manual/mc-manual.html>

Micro Focus. (n.d.). *Fortify static code analyzer documentation*. Micro Focus. Retrieved October 10, 2024, from https://www.microfocus.com/documentation/fortify-static-code-analyzer/

LLVM Project. (n.d.). *Clang Static Analyzer checks*. Clang.llvm.org. Retrieved October 10, 2024, from https://clang-analyzer.llvm.org/available\_checks.html#core-null

OWASP. (n.d.). *OWASP ZAP SQL injection*. OWASP Foundation. Retrieved October 10, 2024, from <https://www.zaproxy.org/docs/alerts/2/>

LLVM Project. (n.d.). *AddressSanitizer documentation*. Clang.llvm.org. Retrieved October 10, 2024, from <https://clang.llvm.org/docs/AddressSanitizer.html>

Cppcheck Developers. (n.d.). *Cppcheck: A tool for static C/C++ code analysis*. Cppcheck. Retrieved October 10, 2024, from <http://cppcheck.sourceforge.net/>