**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 9](#_Toc52464062)

[Coding Standard 5 10](#_Toc52464063)

[Coding Standard 6 11](#_Toc52464064)

[Coding Standard 7 12](#_Toc52464065)

[Coding Standard 8 14](#_Toc52464066)

[Coding Standard 9 15](#_Toc52464067)

[Coding Standard 10 16](#_Toc52464068)

[Defense-in-Depth Illustration 18](#_Toc52464069)

[Project One 18](#_Toc52464070)

[1. Revise the C/C++ Standards 18](#_Toc52464071)

[2. Risk Assessment 18](#_Toc52464072)

[3. Automated Detection 18](#_Toc52464073)

[4. Automation 18](#_Toc52464074)

[5. Summary of Risk Assessments 19](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 19](#_Toc52464076)

[7. Map the Principles 20](#_Toc52464077)

[Audit Controls and Management 21](#_Toc52464078)

[Enforcement 21](#_Toc52464079)

[Exceptions Process 21](#_Toc52464080)

[Distribution 22](#_Toc52464081)

[Policy Change Control 22](#_Toc52464082)

[Policy Version History 22](#_Toc52464083)

[Appendix A Lookups 22](#_Toc52464084)

[Approved C/C++ Language Acronyms 22](#_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 data validation ensures that all data entering a system is verified and adheres to expected formats and values. Proper validation prevents attackers from injecting malicious code and exploiting vulnerabilities such as SQL injection, buffer overflow, and cross-site scripting attacks, thus preserving the integrity and security of applications. |
| 1. Heed Compiler Warnings | Paying attention to compiler warnings means addressing and resolving these warnings promptly instead of ignoring them. Compiler warnings often highlight potential weaknesses, flaws, or security risks in the code that, if left unaddressed, can develop into serious security vulnerabilities or application defects. |
| 1. Architect and Design for Security Policies | Designing for security from the outset involves integrating security considerations into the software architecture and planning stages. Clearly defined and consistently applied security policies ensure that systems have robust protections built into their foundations, making it easier to enforce, manage, and update security measures throughout the software lifecycle. |
| 1. Keep It Simple | The principle of simplicity emphasizes creating clear and straightforward software designs and implementations. Simple systems are easier to understand, test, audit, and maintain, significantly reducing the likelihood of undetected vulnerabilities and simplifying the security review process. |
| 1. Default Deny | The default deny strategy is a defensive security approach where all access and permissions are initially denied unless explicitly granted. Minimizing exposure and strictly controlling access helps ensure that unauthorized or unexpected behavior does not compromise system security. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege requires granting users, applications, and processes only the minimum level of access necessary to perform their intended functions. By restricting privileges in this manner, the potential damage if an account or application is compromised is significantly limited, reducing overall risk. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization involves cleansing or transforming data before it is transmitted to other systems, ensuring the removal or neutralization of potentially harmful inputs. Proper sanitization helps prevent vulnerabilities such as injection attacks, maintaining the reliability and security of data exchanges between interconnected systems. |
| 1. Practice Defense in Depth | Defense in depth employs multiple security measures to protect systems and data. This layered approach ensures that if one security mechanism fails or is bypassed, other defenses remain effective, increasing overall resilience against cyber threats and reducing single points of failure. |
| 1. Use Effective Quality Assurance Techniques | Adequate quality assurance includes comprehensive testing and verification processes such as automated testing, manual reviews, and code inspections. By rigorously assessing software, developers can identify and fix vulnerabilities early, improving the overall security and reliability of the application. |
| 1. Adopt a Secure Coding Standard | Adopting secure coding standards involves consistently applying clearly defined rules and guidelines for secure software development practices. These standards guide developers toward best practices, prevent common vulnerabilities, and ensure safe, readable, and maintainable code, ultimately enhancing the long-term security posture of software applications. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Correct data type usage prevents overflow errors, data loss, and unexpected behaviors. Proper selection and matching data types ensure predictable program execution and maintain data integrity. |

| **Noncompliant Code** |
| --- |
| The example below improperly mixes signed and unsigned integers, leading to unintended comparison outcomes due to implicit type conversions. |
| unsigned int a = 10;  int b = -5;  if (a < b) {  /\* unexpected behavior due to implicit conversion \*/  } |

| **Compliant Code** |
| --- |
| Properly match data types to avoid implicit type conversions and ensure logical correctness. |
| int a = 10;  int b = -5;  if (a < b) {  /\* correct behavior with matching signed types \*/  } |

| **Principles(s):**  **Heed Compiler Warnings** — Compiler warnings can reveal unsafe type conversions and potential overflows; addressing them helps prevent subtle data type vulnerabilities. **Validate Input Data** — Validating data ensures that only correctly typed and properly formatted input is processed, reducing the risk of logic errors or data corruption. **Adopt a Secure Coding Standard** — Enforcing clear rules for correct data type usage prevents common coding errors that can compromise program stability and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Integer-overflow | Fully checked |
| CodeSonar | 9.0p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Parasoft C/C++test | 2024.2 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid signed integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| Helix QAC | 2024.4 | C2800, C2860  C++2800, C++2860  DF2801, DF2802, DF2803, DF2861, DF2862, DF2863 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validating data values ensures data integrity and prevents exploitation through malicious or erroneous inputs. Data validation serves as the first line of defense against injection attacks and corrupt states. |

| **Noncompliant Code** |
| --- |
| The following example fails to validate user-provided input, risking invalid or harmful data usage. |
| int age;  cin >> age; |

| **Compliant Code** |
| --- |
| Validate input explicitly to ensure it falls within acceptable ranges before usage. |
| int age;  cin >> age;  if(age < 0 || age > 130) {  throw std::invalid\_argument("Invalid age");  } |

| **Principles(s):**  **Validate Input Data** — Verifying that input values fall within acceptable ranges prevents invalid or malicious data from causing incorrect program behavior or security issues. **Sanitize Data Sent to Other Systems** — Properly validating and sanitizing data before transmitting it to other systems prevents injection attacks and ensures data integrity. **Adopt a Secure Coding Standard** — Consistently enforcing limits on untrusted integer values ensures predictable and safe operation of the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by taint analysis |
| Helix QAC | 2024.4 | DF2794, DF2804, DF2854, DF2859, DF2864, DF2894, DF2899, DF2904, DF2909, DF2914, DF2924, DF2944, DF2949, DF2954, DF2956, DF2959 |  |
| Parasoft C/C++test | 2024.2 | CERT\_C-INT04-a  CERT\_C-INT04-b  CERT\_C-INT04-c | Avoid potential integer overflow/underflow on tainted data Avoid buffer read overflow from tainted data Avoid buffer write overflow from tainted data |
| Polyspace Bug Finder | R2024b | CERT C: Rec. INT04-C | Checks for:   * Array access with tainted index * Loop bounded with tainted value * Memory allocation with tainted size * Tainted size of variable length array   Rec. partially supported. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Proper handling of strings prevents buffer overflow vulnerabilities, memory corruption, and ensures data integrity during string operations. |

| **Noncompliant Code** |
| --- |
| Unchecked usage of strcpy() can result in buffer overflows if the source string exceeds buffer size. |
| char buf[10];  strcpy(buf, input); |

| **Compliant Code** |
| --- |
| Employ safer string functions such as strncpy() to limit the number of characters copied and explicitly terminate strings. |
| char buf[10];  strncpy(buf, input, sizeof(buf) - 1);  buf[sizeof(buf) - 1] = '\0'; |

| **Principles(s):**  **Validate Input Data** — Validating string input helps ensure that only properly sized and formatted data is processed, reducing the risk of buffer overflows. **Adopt a Secure Coding Standard** — Using bounded string functions aligns with secure coding practices and prevents common vulnerabilities like memory corruption and buffer overflows. **Use Effective Quality Assurance Techniques** — Testing string-handling code with both typical and edge-case inputs helps detect unsafe operations and verify correctness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p9 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |
| Polyspace Bug Finder | R2024b | CERT C++: STR50-CPP | Checks for:  Use of dangerous standard function  Missing null in string array  Buffer overflow from incorrect string format specifier  Destination buffer overflow in string manipulation  Insufficient destination buffer size  Rule partially covered. |
| RuleChecker | 22.10 | stream-input-char-array | Partially checked |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Protecting databases from SQL injections ensures data confidentiality, integrity, and prevents unauthorized access or manipulation. SQL injection attacks occur when untrusted input is improperly incorporated into database queries, allowing attackers to execute arbitrary SQL commands. Using parameterized queries eliminates this risk by separating data from code. |

| **Noncompliant Code** |
| --- |
| Directly concatenating untrusted user input into an SQL query string allows attackers to craft malicious inputs that modify query behavior, enabling SQL injection attacks. |
| std::string user\_input;  std::cin >> user\_input;  std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "'";  executeQuery(query); |

| **Compliant Code** |
| --- |
| Using parameterized queries or prepared statements ensures that user input is treated strictly as data and not executable SQL code, preventing injection attacks. |
| std::string user\_input;  std::cin >> user\_input;  std::unique\_ptr<sql::PreparedStatement> stmt(  connection->prepareStatement("SELECT \* FROM users WHERE username = ?"));  stmt->setString(1, user\_input);  std::unique\_ptr<sql::ResultSet> res(stmt->executeQuery()); |

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

| **Principles(s):**  **Validate Input Data** — Validating and controlling user input reduces the risk of injection of malicious SQL commands. **Sanitize Data Sent to Other Systems** — Ensuring that input is properly parameterized and sanitized before sending it to the database protects against injection attacks. **Adopt a Secure Coding Standard** — Using secure database APIs and parameterized queries follows best practices for preventing SQL injection vulnerabilities. **Adhere to the Principle of Least Privilege** — Database accounts used by the application should only have the minimum permissions necessary, limiting the impact of any successful injection attack. **Practice Defense in Depth** — Layering controls such as input validation, prepared statements, strict database permissions, and monitoring reduces the likelihood and potential impact of SQL injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 9.0p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Parasoft C/C++test | 2024.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection Protect against file name injection Protect against SQL injection |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Ensuring proper memory management protects against memory leaks, buffer overflows, and stability issues, ultimately preserving application reliability. |

| **Noncompliant Code** |
| --- |
| Memory allocation without explicit checks for successful allocation can cause crashes. |
| int \*data = (int\*)malloc(10 \* sizeof(int)); |

| **Compliant Code** |
| --- |
| Always verify successful memory allocation to avoid undefined behavior. |
| int \*data = (int\*)malloc(10 \* sizeof(int));  if (!data) {  throw std::bad\_alloc();  } |

| **Principles(s):**  **Heed Compiler Warnings** — Many compilers issue warnings about potential memory leaks, uninitialized pointers, or dangerous memory operations that should be addressed early. **Practice Defense in Depth** — Proper memory management adds a layer of protection against exploitation of memory corruption vulnerabilities. **Adopt a Secure Coding Standard** — Following clear rules for memory management ensures predictable behavior and prevents resource leaks. **Architect and Design for Security Policies** — Resource management should be considered during system design to ensure reliable, maintainable, and secure memory handling throughout the application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2025.1 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| Klocwork | 2025.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Polyspace Bug Finder | R2024b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Using assertions helps validate assumptions and detect logic errors early, facilitating easier debugging and enhancing overall robustness. |

| **Noncompliant Code** |
| --- |
| Missing assertion checks can cause runtime errors if unexpected conditions occur. |
| int divisor = getDivisor();  int result = 10 / divisor; |

| **Compliant Code** |
| --- |
| Explicitly validate critical assumptions using assertions. |
| int divisor = getDivisor();  assert(divisor != 0 && "Divisor should never be zero");  int result = 10 / divisor; |

| **Principles(s):**  **Heed Compiler Warnings** — Compiler warnings may highlight improper or unsafe use of assertions, helping developers enforce correct usage. **Architect and Design for Security Policies** — Assertions should be part of a planned architecture for defensive programming, not used for runtime validation. **Adopt a Secure Coding Standard** — Using assertions appropriately ensures they are used to catch programming errors, not to enforce runtime behavior, aligning with secure coding best practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2024.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling ensures that software can respond gracefully to unexpected conditions without crashing or leaving the system in an unstable state. Unhandled exceptions can cause application termination, while overly generic or silent catch blocks can obscure real problems and make debugging or auditing difficult. |

| **Noncompliant Code** |
| --- |
| This example uses a generic catch-all handler (catch(...)) without logging or corrective action, which hides critical errors. This practice, known as error hiding, makes debugging difficult and can allow serious faults to go unnoticed. |
| void processUserRequest() {  try {  // Code that may throw exceptions  performSensitiveOperation();  } catch (...) {  // Error hiding: no logging, no rethrow, no user feedback  // The program silently fails  }  } |

| **Compliant Code** |
| --- |
| This example catches specific exceptions and takes appropriate action, such as logging the error or safely notifying the user. Catch-all is used only as a last resort and includes logging to avoid silent failures. |
| #include <iostream>  #include <stdexcept>  void processUserRequest() {  try {  performSensitiveOperation();  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what() << std::endl;  } catch (const std::exception& e) {  std::cerr << "General exception: " << e.what() << std::endl;  } catch (...) {  std::cerr << "Unknown error occurred during user request." << std::endl;  // Catch-all used with caution and visibility  }  } |

| **Principles(s):**  **Heed Compiler Warnings** — Compiler warnings can reveal unhandled exceptions or unsafe exception usage that should be corrected. **Adopt a Secure Coding Standard** — Following best practices for exception handling ensures predictable and secure application behavior during error conditions. **Use Effective Quality Assurance Techniques** — Thorough testing of exception handling paths ensures that applications fail gracefully and maintain integrity when errors occur. **Practice Defense in Depth** — Proper exception handling adds an additional layer of resilience by ensuring that unexpected errors do not lead to system compromise or data loss. **Keep It Simple** — Exception handling code should be clear, predictable, and easy to maintain to avoid introducing additional complexity or errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2024b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-008-CPP | Proper resource management ensures that system resources such as memory, file handles, sockets, and other critical resources are allocated and released correctly. Failure to release resources can lead to memory leaks, resource exhaustion, degraded system performance, and denial of service. Adhering to consistent patterns of resource cleanup is essential for system reliability and security. |

| **Noncompliant Code** |
| --- |
| Allocating dynamic memory without corresponding deallocation results in a memory leak. Over time, repeated allocations without proper cleanup can exhaust system memory and degrade performance or cause program crashes. |
| int\* ptr = new int[10];  // No delete[] call — memory leak occurs if this pointer is not released. |

| **Compliant Code** |
| --- |
| Releasing allocated resources when they are no longer needed ensures system resources are available and prevents memory leaks. Additionally, setting the pointer to nullptr after deletion avoids dangling pointers and accidental reuse. |
| int\* ptr = new int[10];  // Use the allocated memory as needed...  delete[] ptr;  ptr = nullptr; // Prevent dangling pointer use |

| **Principles(s):**  **Heed Compiler Warnings** — Compiler warnings about resource leaks or misuse help identify potential resource management issues. **Practice Defense in Depth** — Proper resource management prevents resource exhaustion attacks and ensures system resilience. **Adopt a Secure Coding Standard** — Following consistent patterns for allocating and releasing resources prevents leaks and ensures predictable system behavior. **Architect and Design for Security Policies** — Resource management should be incorporated into system architecture, using patterns such as smart pointers to automate safe handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Double free  Type mismatch  Leak |
| Parasoft C/C++tes | 2024.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets ([]) for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resources |
| Polyspace Bug Finder | R2024b | CERT C++: MEM51-CPP | Checks for:  Invalid deletion of pointer  Invalid free of pointer  Deallocation of previously deallocated pointer  Rule partially covered. |
| Helix QAC | 2025.1 | C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | STD-009-CPP | Preventing misuse of pointers helps avoid segmentation faults, undefined behavior, and memory corruption. |

| **Noncompliant Code** |
| --- |
| Dereferencing uninitialized pointers leads to unpredictable program behavior. |
| int\* ptr;  \*ptr = 10; |

| **Compliant Code** |
| --- |
| Always initialize pointers before dereferencing to maintain safe and predictable program behavior. |
| int value = 10;  int\* ptr = &value;  \*ptr = 20; |

| **Principles(s):**  **Heed Compiler Warnings** — Compiler warnings about uninitialized or dangling pointers help detect unsafe pointer usage. **Adopt a Secure Coding Standard** — Using safe pointer practices prevents memory corruption and undefined behavior. **Use Effective Quality Assurance Techniques** — Tools such as static analyzers and memory sanitizers help detect unsafe pointer operations. **Practice Defense in Depth** — Ensuring pointer safety reduces the risk of exploitation through memory corruption or buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | uninitialized-read | Partially checked |
| CodeSonar | 9.0p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Polyspace Bug Finder | R2024b | CERT C++: EXP53-CPP | Checks for:  Non-initialized variable  Non-initialized pointer  Rule partially covered. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency Safety | STD-010-CPP | Concurrency errors such as race conditions, deadlocks, and improper synchronization can lead to unpredictable and unsafe behavior in multithreaded applications. Race conditions occur when multiple threads access shared data simultaneously without proper synchronization, potentially corrupting data or causing crashes. Using appropriate synchronization mechanisms ensures correctness and reliability in concurrent code. |

| **Noncompliant Code** |
| --- |
| This example demonstrates a race condition where two threads increment a shared variable without synchronization. The result is nondeterministic and may corrupt the final value. |
| #include <thread>  #include <iostream>  int count = 0;  void increment() {  for (int i = 0; i < 1000; ++i) {  count++; // Race condition  }  }  int main() {  std::thread t1(increment);  std::thread t2(increment);  t1.join();  t2.join();  std::cout << "Final count: " << count << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using a mutex ensures that increments to the shared variable are performed atomically, eliminating the race condition. |
| #include <thread>  #include <mutex>  #include <iostream>  int count = 0;  std::mutex mtx;  void increment() {  for (int i = 0; i < 1000; ++i) {  std::lock\_guard<std::mutex> lock(mtx);  count++;  }  }  int main() {  std::thread t1(increment);  std::thread t2(increment);  t1.join();  t2.join();  std::cout << "Final count: " << count << std::endl;  return 0;  }});  std::thread t2([&]() { std::lock\_guard<std::mutex> lock(mtx); count++; }); |

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

| **Principles(s):**  **Practice Defense in Depth** — Proper use of synchronization primitives adds layers of protection against concurrency-related vulnerabilities such as race conditions and deadlocks. **Adopt a Secure Coding Standard** — Applying secure concurrency patterns ensures that multithreaded code behaves predictably and safely under all conditions. **Architect and Design for Security Policies** — Concurrency control must be planned at the architecture level to avoid introducing subtle and hard-to-detect synchronization errors. **Keep It Simple** — Using clear, well-structured synchronization mechanisms reduces complexity and prevents errors in concurrent code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | read\_write\_data\_race  write\_write\_data\_race | Supported |
| CodeSonar | 9.0p0 | CONCURRENCY.DATARACE | Data Race |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CON52-a | Use locks to prevent race conditions when modifying bit fields |
| Helix QAC | 2025.1 | C++1774, C++1775 |  |

### 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 integrated throughout the Green Pace DevSecOps pipeline to ensure consistent enforcement of coding standards and secure development practices. In the pre-production phase, automation should begin during the Assess and Plan and Design stages. This can be achieved using tools that automatically scan threat landscapes, validate architectural compliance with OWASP standards, and enforce secure design patterns. In the Build stage, static code analysis tools such as SonarQube can continuously scan for violations of C/C++ coding standards outlined in this policy. As the code is developed, automated testing frameworks will ensure functional correctness and compliance by incorporating security-focused unit tests and vulnerability scanners before the application progresses to deployment.

In the production phase, automation should continue during the Monitor and Detect and Maintain and Stabilize stages. Automated tools will be used to monitor log files, detect intrusion attempts, and assess runtime behavior against known baselines. In the event of any anomalies or deviations, automated response systems can initiate rollback procedures, disable vulnerable services, or notify security personnel. This continuous feedback loop not only ensures the rapid mitigation of issues but also helps enforce secure behavior in real time. By embedding automation across both development and operations, Green Pace can ensure continuous compliance with its secure coding standards while maintaining a high velocity and responsiveness in its software lifecycle.

### 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 | Medium | P12 | L1 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | High | P9 | L2 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | Low | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| STD-009-CPP | High | Probable | Medium | P12 | L1 |
| STD-0010-CPP | Medium | Probable | Medium | P8 | 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 is used to protect stored data on physical or virtual media, such as databases, file systems, and backup storage. This protection is achieved through encryption algorithms, such as AES-256, which render the data unreadable without the corresponding decryption keys. This process applies to stored credentials, customer data, and any other sensitive assets. It ensures that even if the storage medium is compromised, the data remains secure. |
| Encryption in flight | Encryption in flight ensures that data remains secure while it travels across networks between clients, services, and systems. This security is achieved using protocols such as TLS, HTTPS, SSL, or other like protocols, which help prevent data interception or tampering during transmission. These measures apply to all communication channels, protecting against threats like man-in-the-middle attacks and unauthorized snooping during data transfers. |
| Encryption in use | Encryption in use protects data while it is actively being processed in memory. It is implemented using secure enclaves, memory encryption, or isolated execution environments to prevent exposure from memory scraping or runtime attacks. This approach is particularly important during sensitive operations, as it reduces the risks associated with insider threats or compromised processes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user's or system's identity before granting access to resources. This process is enforced using various mechanisms, including multi-factor authentication, tokens, and digital certificates. It occurs during every login or access request to ensure that only authorized individuals can gain entry to protected systems. |
| Authorization | Authorization determines the actions that a user or system is allowed to take after they have been successfully authenticated. It can be implemented using methods such as role-based access control (RBAC), access control lists (ACLs), or attribute-based rules. This process applies to all resource access, including defining user levels of access and enforcing the principle of least privilege to prevent unauthorized actions. |
| Accounting | Accounting involves logging, monitoring, and auditing of user and system activities. This process is carried out through centralized logging systems, audit trails, and access reports that track actions such as login, file access, modifications, database changes, and the addition or removal of user accounts. In all environments, it is essential to ensure accountability, detect anomalies, and support compliance requirements. |

**\***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 | 06/14/2025 | Completed Template | Austin Sarkis |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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