**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 in C and C++ is essential to prevent problems like buffer overflow. "Secure Coding in C and C++ (Second Edition)" recommends input validation as the best defense against such issues. Follow SEI CERT C++ Rule 07 for proper input handling using functions like fputs() or fprints(). |
| 1. Heed Compiler Warnings | Validating code for buffer overflow is crucial. Pay attention to compiler warnings during the build. Use tools like Clang-Tidy and Cppcheck to find build errors. Prioritize enabling compiler warnings as the first defense. Visual Studio C++ compiler helps catch issues like uninitialized variables, format string mismatches, and using functions before declaring them. Enabling basic runtime checks (/RTCs) in C/C++ options helps identify buffer overflows with minimal performance impact. |
| 1. Architect and Design for Security Policies | Start security measures early in software development. Educate users, apply least privilege, and segregate networks. Divide the system into separate parts with specific privileges for secure operations. Ensure data sanitation, input validation, and a deny-by-default approach. Share algorithms for security review. Stay updated on vulnerabilities through NIST, OWASP, CERT, and academic research. |
| 1. Keep It Simple | Simplify your code for clarity, efficiency, and effectiveness. Use descriptive names for variables and purposeful names for methods. Employ comments to explain code. Make sure each Class has a single responsibility and remove any unused instances, methods, or global states. KISS (Keep It Simple, Stupid) is achieved when efficiency remains intact after following these rules. |
| 1. Default Deny | Deny permissions by default, allowing access only when explicitly authorized. For instance, firewalls block unpermitted traffic. SQL Server permits only added users with EXECUTE permissions. Using nonstandard ports adds security. Fail2Ban bans users after incorrect password attempts. File permissions in Unix systems control access with specific settings for reading or writing, usually managed by root or user permissions. |
| 1. Adhere to the Principle of Least Privilege | Limit user accounts to minimal privileges, restricting capabilities beyond basic tasks. Avoid widespread Root user accounts. MYSQL uses unique accounts for specific DB tasks, reducing attack surfaces and malicious actions. In C, 'const' ensures read-only privileges for class members during function execution. Avoid passing unnecessary parameters to methods. |
| 1. Sanitize Data Sent to Other Systems | Preventing vulnerabilities in a program involves accepting only valid inputs and managing all possible valid combinations. Avoiding strings that trigger commands or cause buffer overflow is crucial. White-listing, which allows only valid inputs (deny by default), ensures safety against unexpected inputs. It's a preferred method as predicting all invalid values for black-listing can be tough. Data sanitation also includes parameterizing SQL queries to prevent SQL injection. |
| 1. Practice Defense in Depth | Defense in Depth uses multiple independent defense layers. IT and server maintenance employ various tools like firewalls, intrusion detection, antivirus software, VPNs, and secure configurations. At a programmatic level, guard against attacks like XML and SQL injection, validate user input rigorously, and use secure encryption protocols to avoid costly vulnerabilities. Rely on established community libraries and trusted crypto code for better security instead of creating your own solutions. |
| 1. Use Effective Quality Assurance Techniques | Use Tidy-Clang in Visual Studio C++ and cppcheck for static analysis to fix programming errors. Integrate Coverity Scan for further checks in continuous integration workflows. Implement Exceptions for critical function failures and assertions to test conditions in development. Ensure adherence to user requirements using SCRUM or AGILE approaches for quality assurance. |
| 1. Adopt a Secure Coding Standard | For Quality Assurance and the Software Development Lifecycle (SDLC), adopt the SEI CERT C++ Coding Standard for secure coding in C++. It covers various aspects like declarations, memory management, input/output, and more. These principles can guide the continuous integration process. |

### 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] | When selecting a data type, remember that unsigned integer values shouldn't become negative, while signed integers are suitable for values that can be negative. It's advisable to use the smallest signed or unsigned type that can fully represent the potential values for the variable, which helps conserve memory. For instance, using short total = strlen(argv[1]) + 1; might permit negative sizes and lack sufficient range, whereas size\_t total = strlen(argv[1]) + 1; employs size\_t, offering more precise representation.  Guideline: INT32-C. Ensure that operations on signed integers avoid overflow. |

| **Noncompliant Code** |
| --- |
| The code doesn't follow the guideline, using 'short' for string length storage, risking inaccuracies and negative sizes, instead of the recommended 'size\_t' data type. |
| int main(int argc, char\* argv[]) {  short total = strlen(argv[1]) + 1; // Using 'short' for total size  std::cout << "Total length: " << total << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The code is compliant as it uses the 'size\_t' data type to store the length of the string (strlen(argv[1]) + 1). Using 'size\_t' ensures a suitable representation for the potential size of the string, aligning with the guideline to use appropriate data types for accurate storage of string lengths without risking negative sizes. |
| int main(int argc, char\* argv[]) {  size\_t total = strlen(argv[1]) + 1; // Using 'size\_t' for total size  std::cout << "Total length: " << total << std::endl;  return 0;  } |

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

| **Principles(s):** The principle related to this standard is INT32-C, aiming to prevent overflow in operations with signed integers. The standard advocates choosing suitable data types to accurately represent variable ranges. In the compliant code, using 'size\_t' aligns with INT32-C by preventing potential overflow and accurately representing larger string lengths, ensuring safe operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Helix QAC | 2021.2 | C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863  C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863 | Implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| TrustInSoft Analyzer | 1.38 | Signed\_overflow | Exhaustively verified |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-ccp] | Utilize the SEI CERT C++ Coding Standard in both Quality Assurance and the Software Development Lifecycle (SDLC) to ensure secure coding in C++. It addresses declarations, memory management, input/output, and other critical aspects, providing guidance for the continuous integration process. |

| **Noncompliant Code** |
| --- |
| The code is noncompliant as it assigns a negative value to an unsigned integer variable, violating the standard that expects unsigned variables to hold only non-negative values, potentially causing unexpected behavior due to unsigned integer wraparound effects. |
| int main() {  int negativeValue = -10;  unsigned int count = negativeValue;  std::cout << "Count: " << count << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The code is compliant as it assigns a positive value to an unsigned integer variable, following the standard that expects unsigned variables to hold only non-negative values, ensuring expected behavior without potential wraparound effects in unsigned integer types. |
| int main() {  int positiveValue = 10;  unsigned int count = positiveValue;  std::cout << "Count: " << count << std::endl;  return 0;  } |

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

| **Principles(s):** 1) Verify input data at the start to identify errors early on, and 2) Employ efficient quality assurance methods using analysis tools to address these issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |
| CodeSonar | 6.1p0 | 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 | 2021.1 | CERT\_C-INT30-a  CERT\_C-INT30-b  CERT\_C-INT30-c | Avoid integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| Polyspace Bug Finder | R2021a | CERT C: Rule INT30-C | Checks for:  Unsigned integer overflow  Unsigned integer constant overflow  Rule partially covered. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-ccp] | Incorrect string sizes and overlooked buffer boundaries may cause buffer overflows and runtime errors. Avoid copying unbounded data like stdin to fixed-length arrays. Follow guidelines like STR30-C and STR31-C for safer string handling. Be cautious with UTF-decoders (MSC10-C) as they can lead to security issues. A secure model is the "callee allocates, callee frees" approach in C++'s std::basic\_string template, preventing costly mistakes without requiring null terminators. Avoid using `gets()` due to its insecurity. |

| **Noncompliant Code** |
| --- |
| The code is noncompliant as it copies unbounded data from `stdin` to a fixed-length array (`dest`), potentially causing buffer overflow due to inadequate space allocation and violating the coding standard for data values. |
| int main() {  char dest[5];  std::cin >> dest;  return 0;  } |

| **Compliant Code** |
| --- |
| Compliant code uses `std::string` for dynamic storage of input, meeting the coding standard by avoiding fixed-length arrays and potential buffer overflow issues associated with inadequate space allocation. |
| int main() {  std::string input;  std::cin >> input;  return 0;  } |

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

| **Principles(s):** 1) Verify input using suitable string functions, 2) Pay attention to compiler warnings about overflow, 3) Cleanse data to avoid string attacks, and 4) Utilize Quality Assurance techniques in the CI/CD pipeline to detect these errors during the build process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-cpp] | String-based attack vectors, such as command-line arguments, environmental variables, console input, text files, and network connections, provide openings for strategic SQL injection attacks, causing potential overflow. As per Secure Coding in C and C++, string concatenation serves as a primary point for script injection. To mitigate these risks, validate input size and data type, impose appropriate limits, reject entries containing binary data, escape sequences, and comment characters. Review code invoking EXECUTE, EXEC, or sp\_executesql, and consistently employ Parameterized Queries in SQL.  In C++, the operator >> extracts characters into an array, stopping at valid white spaces, null characters, or EOF. To prevent buffer overflow, restrict the extraction by setting the field width (ios\_base::width or setw()) to a value greater than 0, aligning with the SEI Rule STR31-C to ensure adequate space for strings. |

| **Noncompliant Code** |
| --- |
| The code is noncompliant as it directly copies user input into a fixed-length array without validating the input size, potentially leading to buffer overflow and violating the coding standard. |
| int main() {  char dest[5];  std::cin >> dest;  return 0;  } |

| **Compliant Code** |
| --- |
| The code complies with the standard by utilizing `std::string` for dynamic storage, ensuring input size management and avoiding buffer overflow issues related to fixed-size arrays. |
| int main() {  std::string input;  std::cin >> input;  return 0;  } |

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

| **Principles(s):** 1) Validate input thoroughly and employ query parameterization techniques, 2) Implement secure design principles, considering potential SQL injection threats in the code architecture, 3) Apply a layered defensive strategy, practicing defense in depth to avert such attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | l1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-cpp] | A stack buffer overflow, known as stack smashing, occurs when a security flaw allows an untrusted user to fill the stack buffer with data, leading to potential code injection and control hijacking. Null or invalid pointer dereferencing can result in undefined behavior and segmentation faults. Pointer subterfuge exploits can manipulate function pointers, enabling control transfer to attacker-supplied code. Mitigations involve eliminating buffer overflow risks and storing encrypted pointers. Adherence to CERT C++ security guidelines like MEM50-CPP, EXP53-CPP, MEM52-CPP, and MEM31-CPP is crucial, emphasizing avoiding freed memory access, preventing uninitialized memory reads, handling memory allocation errors, and freeing dynamically allocated memory. Additionally, following MEM08-CPP from the CERT C++ Coding Standard helps avoid subtle memory issues arising from raw memory allocation and deallocation. |

| **Noncompliant Code** |
| --- |
| The code is noncompliant as it copies input to a fixed-size buffer without checking its length, potentially causing a buffer overflow, violating memory protection standards. |
| void vulnerableFunction(const char\* input) {  char buffer[10];  strcpy(buffer, input);  // ...  }  int main() {  const char\* maliciousInput = "This input is way too long and dangerous";  vulnerableFunction(maliciousInput);  return 0;  } |

| **Compliant Code** |
| --- |
| The code complies with memory protection standards by safely copying input to a fixed-size buffer, avoiding buffer overflow vulnerabilities as specified in memory protection guidelines. |
| void safeFunction(const char\* input) {  const size\_t maxLength = 10;  char buffer[maxLength];    strncpy(buffer, input, maxLength - 1);  buffer[maxLength - 1] = '\0';  // ...  }  int main() {  const char\* safeInput = "Safe input";  safeFunction(safeInput);  return 0;  } |

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

| **Principles(s):** Train the team in secure memory allocation, implement a consistent Secure Coding standard, and use QA techniques in CI/CD scans to address vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | l1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Parasoft C/C++test | 2021.1 | 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 | R2021a | 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-ccp] | As a guiding principle, employ assert statements to detect conditions that should never occur if the code is correct; exceptions should be avoided for such situations as they don't require runtime recovery. Asserts signify errors that need fixing, so using exceptions for these serves little purpose.  Assert statements serve as visual indicators for programmers, effectively identifying deprecated code errors and immediately halting on bad inputs. They're particularly valuable for testing error conditions that should have been handled earlier in the code. However, avoid substituting error-handling code with assertions. For additional details, refer to MSC11-C for diagnostic tests using assertions. |

| **Noncompliant Code** |
| --- |
| The code is noncompliant as it uses the `assert` statement for input validation, contrary to its intended purpose for debugging, potentially leading to runtime errors. |
| void processInput(int value) {  assert(value > 0);  // Process the input...  }  int main() {  int userInput = -5; // Negative input  processInput(userInput);  return 0;  } |

| **Compliant Code** |
| --- |
| The code ensures compliance with the assertions guideline by using conditional checks instead of assert for input validation, providing proper handling of invalid input and preventing potential runtime errors. |
| void processInput(int value) {  if (value <= 0) {  // Handle invalid input here...  std::cout << "Invalid input detected!" << std::endl;  }  else {  // Process the input...  std::cout << "Input processed successfully!" << std::endl;  }  }  int main() {  int userInput = -5; // Negative input  processInput(userInput); // Validate and process input  return 0;  } |

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

| **Principles(s):** Architecting for security includes educating the team in crafting secure memory allocation procedures, implementing a unified Secure Coding standard throughout the project, and employing Quality Assurance Techniques to scan for and mitigate vulnerabilities in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2021.1 | CERT\_C-MSC11-a | CERT\_C-MSC11-a |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-ccp] | In determining when to throw an exception, it is advisable to throw an exception when an issue prevents a function from fulfilling its task, but if the function can recover and provide promised services, it should handle the problem without throwing an exception.  Regarding APIs, if an API encounters an unrecoverable error, it should throw the exception back to the client code, allowing the client to decide on handling options like retrying after a specified time or using an alternative network endpoint. Exceptions are suitable for checking errors in public function parameters.  Always throw exceptions by value and catch them by reference or const reference to ensure proper memory management by the compiler for the exception object. Moreover, it's essential to avoid catching exceptions that cannot be handled.  Relevant SEI Cert C++ Standards:  - ERR61-CPP: Catch exceptions by lvalue reference  - ERR51-CPP: Handle all exceptions  - ERR57-CPP: Prevent resource leaks when handling exceptions  - ERR56-CPP: Ensure exception safety, especially the "strong exception safety guarantee," ensuring no observable effects on program state if an operation terminates by raising an exception. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses a string instead of a proper exception type, violating the guideline by not using standard exception types and handling. It may lead to issues in exception management and does not conform to best practices for exception handling as defined by relevant standards ERR61-CPP, ERR51-CPP, ERR57-CPP, and ERR56-CPP. |
| void processInput(int value) {  if (value <= 0) {  throw "Value should be greater than zero.";  }  // Process the input...  std::cout << "Input processed successfully!" << std::endl;  }  int main() {  try {  int userInput = -5; // Negative input  processInput(userInput); // Validate and process input  } catch (const std::exception& e) {  std::cout << "Exception caught: " << e.what() << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant code uses exceptions by throwing std::invalid\_argument for invalid input conditions, adhering to best practices. It demonstrates proper exception handling by catching exceptions with std::exception and providing error messages, ensuring compliance with guidelines like ERR61-CPP, ERR51-CPP, ERR57-CPP, and ERR56-CPP. |
| void processInput(int value) {  if (value <= 0) {  throw std::invalid\_argument("Value should be greater than zero."); // Throw exception for invalid input  }  // Process the input...  std::cout << "Input processed successfully!" << std::endl;  }  int main() {  try {  int userInput = -5;  processInput(userInput);  } catch (const std::exception& e) {  std::cout << "Exception caught: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Architecting for security entails training the team in implementing assertions in the development codebase, adopting a Secure Coding standard for uniformity across the project, and utilizing Quality Assurance Techniques to execute these assertions during every scan within the CI/CD pipeline, thereby conducting secure regression testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++4035, C++4036, C++4037 |  |
| Parasoft C/C++test | 2021.1 | 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 | R2021a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| OOP | [STD-008-ccp] | Object-Oriented Programming's security isn't inherently superior to procedural or functional programming; instead, it depends on the programmer's practices. Key concerns include securely invoking virtual functions and handling self-copying objects. SEI Cert C++ Coding Standards outline specific Object-Oriented Programming guidelines:   * OOP50-CPP: Avoid invoking virtual functions from constructors or destructors. * OOP51-CPP: Avoid slicing derived objects. * OOP52-CPP: Don't delete a polymorphic object without a virtual destructor. * OOP53-CPP: Arrange constructor member initializers in the canonical order. * OOP54-CPP: Manage self-copy assignment gracefully. * OOP55-CPP: Avoid using pointer-to-member operators to access nonexistent members. * OOP56-CPP: Adhere to replacement handler requirements. * OOP57-CPP: Prefer special member functions and overloaded operators over C Standard Library functions. * OOP58-CPP: Ensure copy operations do not mutate the source object. |

| **Noncompliant Code** |
| --- |
| The provided code violates the OOP50-CPP guideline by invoking a virtual function (performAction()) from the constructor of the Base class. This practice is discouraged as it can lead to unexpected behavior in object initialization and violates Object-Oriented Programming principles. |
| class Base {  public:  Base() {  performAction();  }  virtual void performAction() {  std::cout << "Base performAction()" << std::endl;  }  };  class Derived : public Base {  public:  Derived() {}  void performAction() override {  std::cout << "Derived performAction()" << std::endl;  }  };  int main() {  Derived obj;  return 0;  } |

| **Compliant Code** |
| --- |
| This code complies with Object-Oriented Programming guidelines by avoiding invoking virtual functions from constructors, satisfying the OOP50-CPP guideline. Instead, the virtual function performAction() is called outside the constructors, ensuring proper behavior in object initialization and following best practices in OOP. |
| class Base {  public:  Base() {}  virtual ~Base() {}  virtual void performAction() {  std::cout << "Base performAction()" << std::endl;  }  };  class Derived : public Base {  public:  Derived() {}  void performAction() override {  std::cout << "Derived performAction()" << std::endl;  }  };  int main() {  Derived obj;  obj.performAction();  return 0;  } |

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

| **Principles(s):** Architecting for security encompasses team training in Secure Object-Oriented Programming, adopting the SEI Cert OOP compliant practices, emphasizing simplicity (KISS principle) over complexity, and implementing the principle of least privilege by appropriately applying encapsulation and scope. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | overflow\_upon\_dereference  invalid\_function\_pointer |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP55 |  |
| Helix QAC | 2021.2 | C++2810, C++2811, C++2812, C++2813, C++2814 |  |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-OOP55-a | A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| expressions | [STD-009-ccp] | Ensuring security with APIs beyond the C++ standard library involves staying updated on potential vulnerabilities and exploits. Open-source libraries benefit from collective developer scrutiny, but vulnerabilities can still exist. Wired defines "fuzzing" as systematically testing software for weaknesses by trying various data permutations. An article on Security Boulevard exemplifies assessing Open Source libraries like FreeImage, highlighting criteria such as build process, accepted inputs, language, usage in projects, previous fuzzing attempts, and quality assurance mechanisms, stressing the importance of thorough evaluation to detect and mitigate security risks. |

| **Noncompliant Code** |
| --- |
| This C++ code showcases improper expression usage by unnecessarily encapsulating the arithmetic operation (b \* 2) in parentheses, making the expression less readable and more complex than necessary for this simple calculation. |
| int main() {  int a = 10;  int b = 5;  int result = a + (b \* 2);  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This C++ code demonstrates the usage of expressions to perform arithmetic operations (in this case, addition and multiplication) and store the result in the result variable. It showcases the ability to manipulate and compute values using expressions within a C++ program. |
| int main() {  int a = 10;  int b = 5;  int result = a + b \* 2;  std::cout << "Result: " << result << std::endl;  return 0;  } |

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

| **Principles(s):** This principle emphasizes crafting clear and straightforward expressions in code, aiming to maintain readability and reduce complexity for improved comprehension and maintenance of the codebase in C++. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<> |
| CodeSonar | |  |  | | --- | --- | |  | 6.1p0 | | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | CERT\_CPP-EXP54-c    Do not use resources that have been freed  The address of an object with automatic storage shall not be returned from a function  The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist |
| Polyspace Bug Finder | R2021a | CERT C++: EXP54-CPP | Checks for:  Non-initialized variable or pointer  Use of previously freed pointer  Pointer or reference to stack variable leaving scope  Accessing object with temporary lifetime  Rule partially covered. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-nnn-LLL] | [Rationalize the standard.] |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| [Noncompliant code block; code should be indented using 12-point Courier New font.] |

| **Compliant Code** |
| --- |
| [Compliant description] |
| [Compliant code block; code should be indented using 12-point Courier New font.] |

**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** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

[Insert your written explanations here.]

### 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 | High | P9 | L2 |
| Std-002-ccp | High | Likely | High | P9 | L2 |
| Std-003-ccp | High | Likely | Medium | P18 | L1 |
| Std-004-ccp | High | Likely | Medium | P18 | L1 |
| Std-005-ccp | High | Likely | Medium | P18 | L1 |
| Std-006-ccp | Low | Unlikely | High | P1 | L3 |
| Std-007-ccp | Low | Probable | Medium | P4 | L3 |
| Std-008-ccp | High | Probable | High | P6 | L2 |
| Std-009-ccp | High | Probable | High | 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 in rest | We will employ full-disk encryption at the server level and implement database encryption using MySQL Server to secure data at rest, alongside establishing a backup strategy. |
| Encryption at flight | Utilize current and secure libraries, implement Public Key infrastructure for end-to-end protection of message bodies or attachments, employ Managed File Transfer or SSH with link expiration, password access, and leverage Data Leak Prevention mechanisms integrated into cloud services. |
| Encryption in use | Utilize identity management mechanisms to verify user roles and identities, enabling conditional access to tool functionality based on user roles and additional parameters. Implement IRM digital rights management to consistently protect documentation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authenticate users by validating their credentials, typically involving usernames and passwords, ensuring secure authentication through a trusted protocol, utilizing either a secured local database or an external AWS server for authentication control. |
| Authorization | Once authentication is complete, the process of authorization will determine the user's permitted access to resources and functionalities, outlining the operations they can execute. |
| Accounting | Track and record user activities during login/logout and resource usage, including user uptime and any configured parameters for comprehensive monitoring. |

**\***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 | 11/30 | 3-2 | Jeffrey Carlson | Jeffrey carlson |
| 1.2 | 11/30 | 6-2 project | Jeffrey Carlson | jeffreycarlson |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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