**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 | The validate input principle refers to the importance of incorporating input validation from all untrusted data sources. The implementation of proper input validation can help prevent software vulnerabilities that are exploited through malicious input. Always suspect malicious activity through external data sources and implement input validation to maintain data integrity and security. |
| 1. Heed Compiler Warnings | Never ignore compiler warnings and utilize the highest warning level capabilities of your compiler. If warnings are found, address the warnings by modifying the code to avoid potential errors and vulnerabilities. It is also important to incorporate static and dynamic analysis tools to find and fix any underlying security flaws. |
| 1. Architect and Design for Security Policies | This principle reinforces the designing of software architecture with the implementation of security policies in mind. Consideration of security policies in the design phase allows for a more foundational implementation of security policies as opposed to an afterthought, leading to a more robust security of the application that is easier to maintain. |
| 1. Keep It Simple | The keep it simple principle involves designing software as simple and as small in scale as possible. Larger and more complex designs have a higher chance that errors will be made which can result in potential vulnerabilities. By adopting a simpler approach in design, it is much easier to implement and maintain security features. |
| 1. Default Deny | Adhering to the default deny principle means incorporating a permission-based system that sets the default permissions to be denied. Access should only be permitted when set privilege conditions are met. This helps to mitigate potential unauthorized access to sensitive data or functionalities of the system. |
| 1. Adhere to the Principle of Least Privilege | Following the principle of least privilege means granting the least privilege necessary to complete a task within the system. If elevated privilege is necessary for the task, only allocate the least amount of time necessary to finish the task. This principle helps to block potential unauthorized access as well as reduce the risk of arbitrary code being executed with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | This principle stresses the importance that all data passed to external systems should be properly sanitized. This includes data sent to command shells, databases, or other components. Improper sanitation of data sent to other systems could lead to injection attacks and potential compromise of sensitive data. |
| 1. Practice Defense in Depth | Practicing defense in depth refers to the practice of applying multiple defensive layers to a project. This is so that if one layer fails to prevent a vulnerability exploit, another layer can potentially catch the issue. Having multiple layers of security can, as a collective, enhance the overall security of a system. |
| 1. Use Effective Quality Assurance Techniques | Ensure the usage of quality assurance techniques as they can help in identifying and addressing vulnerabilities. Use quality assurance practices like penetration testing, fuzz testing, and source code audits as a part of the quality assurance program. Independent security reviews can also be utilized to reinforce security of a system as their singular focus allows for a more thorough inspection on different elements of the code. |
| 1. Adopt a Secure Coding Standard | The adopt a secure coding standard principle means to develop or follow a secure coding standard for the targeted language or platform. By adhering to a secure coding standard during development, best practices designed to mitigate security vulnerabilities will be followed which enhances the overall security of the project. |

### 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** | **Data Type Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | The data type standard verifies that the appropriate data types are utilized for values. Using the correct data types for values can mitigate issues such as integer overflow and underflow. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the incorrect data type of “int” is used for the large\_num variable. This can result in integer overflow as the value being stored is greater than the data type assigned. |
| // Noncompliant example  void process\_data(int data) {    // Do something  }  int main() {  // Incorrect datatype of int for large\_num  int large\_num = 1000000;  process\_data(large\_num);  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, the “long long” data type is used for large\_num. This allows for the data stored in large\_num to be correctly represented and prevents potential integer overflow as seen in the noncompliant example. |
| // Complaint example  void process\_data(long long data) {    // Do something  }  int main() {  // Correct datatype long long for large\_num  long long large\_num = 1000000;  process\_data(large\_num);  return 0;  } |

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

| **Principles(s):** **1, 10.** The Data Type Standard is supported by Principle 1: Validate Input Data and Principle 10: Adopt a Secure Coding Standard. Through the emphasis of proper input validation with principle 1, data type vulnerabilities like integer overflow and underflow can be mitigated. Additionally, following principle 10 and adopting a secure coding standard can further protect against the vulnerabilities associated with using incorrect data types for data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”integerOverflow”  [CWE: 190](https://cwe.mitre.org/data/definitions/190.html) | Analyzing the code with Cppcheck enables the check for “integerOverflow” which can help identify issues with correct data types that result in integer overflow. |
| Clang-Tidy | 19.0.0git | [cppcoreguidelines-pro-type-vararg](https://clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines/pro-type-vararg.html) | Using the clang-tidy checker cppcoreguidelines-pro-type-vararg allows for checking that appropriate data types are used for variable arguments. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | The data value standard ensures that data values are properly validated and sanitized. This helps to mitigate against security risks like buffer overflows and format string vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, user input is not properly validated or sanitized. This can lead to security issues such as buffer overflow as the input can exceed the size of the buffer for user\_input. |
| // Noncompliant example  void process\_input(const char\* input) {  // Data is not validated or sanitized  std::cout << “Input: “ << input << std::endl;  char buffer[10];  std::strcpy(buffer, input);  }  int main() {  char user\_input[20];  std::cout << “Enter input: “;  std::cin >> user\_input;  process\_input(user\_input);  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, measures are introduced for validation and sanitization. Input validation can be observed with the if statement that checks if the input pointer is not null. Also, a buffer is used for bounds checking while snprintf is used to copy the input to the buffer helping to reduce the risk of buffer overflows. Lastly, an else statement communicates an invalid input message to the user if the input is null. |
| // Compliant example  void process\_input(const char\* input) {  // Checks for valid input  if (input != nullptr) {  // Uses a buffer for bounds checking  char buffer[10];  // Uses snprintf to copy input to buffer, mitigating risks of  // buffer overflow  snprintf(buffer, sizeof(buffer), “%s”, input);  std::cout << “Sanitized Input: “ << buffer << std:: endl;  }  else {  // If input is invalid print message to user  std::cerr << “Invalid input” << std::endl;  }  }  int main() {  char user\_input[20];  std::cout << “Enter input: “;  std::cin >> user\_input;  process\_input(user\_input);  return 0;  } |

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

| **Principles(s):** **1, 7.** The Data Value Standard is supported by Principle 1: Validate Input Data and Principle 7: Sanitize Data Sent to Other Systems. By incorporating input validation through principle 1, data values can be properly validated to mitigate against security risks like buffer overflows and format string vulnerabilities. Adhering to principle 7 for sanitizing data sent to other systems will enforce the idea of data sanitation before it is sent to other systems or utilized for critical operations—further protecting against vulnerabilities tied to utilizing improper data values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”bufferAccessOutOfBounds”  [CWE: 788](https://cwe.mitre.org/data/definitions/788.html) | Cppcheck has a checker for accessing memory that is out of bounds that shows the error “bufferAccessOutOfBounds.” This can help discern issues in code with data values that are not properly validated or sanitized. |
| Clang-Tidy | 19.0.0git | [clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling](https://clang.llvm.org/extra/clang-tidy/checks/clang-analyzer/security.insecureAPI.DeprecatedOrUnsafeBufferHandling.html) | Using the clang-tidy clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling checker can detect the misuse of buffer handling functions associated with improper input validation and sanitation. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | The string correctness standard focuses on preventing issues with buffer overflows and null termination. This is accomplished by focusing on the implementation of bounds checking and null termination in string operations. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, strcpy is used without properly checking if the my\_string buffer has enough space. This has the potential risk of a buffer overflow issue if the test\_message string is larger than the my\_string buffer. Additionally, the noncompliant code example doesn’t manually null terminate the string which opens the possibility of string operation issues. |
| // Noncompliant example  int main() {  char my\_string[10];  const char test\_message[] = “Test message!”;    // No bounds checking or null termination  strcpy(my\_string, test\_message);  std::cout << “Message: “ << my\_string << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, snprintf is used with bounds checking to copy the test\_message string to the my\_string variable. This approach prevents buffer overflow and manual null termination is used to further prevent string operation issues. |
| // Compliant example  int main() {  char my\_string[10];  const char test\_message[] = “Test message!”;    // Uses snprintf with bounds checking and ensuring space for null  // termination  snprintf(my\_string, sizeof(my\_string), “%s”, test\_message);  // Null termination  my\_string[sizeof(test\_message) – 1] = ‘\0’;  std::cout << “Message: “ << my\_string << std::endl;  return 0;  } |

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

| **Principles(s):** **1, 8.** The String Correctness Standard is supported by Principle 1: Validate Input Data and Principle 8: Practice Defense in Depth. Enforcing thorough input validation with principle 1 will protect against string correctness vulnerabilities like buffer overflows and null termination issues. Principle 8 and practicing defense in depth will further solidify these secure coding practices by incorporating several layers of defense, ensuring that strings have proper bounds checking and null termination. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”bufferAccessOutOfBounds”  [CWE: 788](https://cwe.mitre.org/data/definitions/788.html) | Like the data values standard, Cppcheck can be used for its checker for accessing memory that is out of bounds that shows the error “bufferAccessOutOfBounds.” This can help discern issues in code with string correctness that causes buffer overflows. |
| Clang-Tidy | 19.0.0git | [clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling](https://clang.llvm.org/extra/clang-tidy/checks/clang-analyzer/security.insecureAPI.DeprecatedOrUnsafeBufferHandling.html) | Using the clang-tidy clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling checker can detect the misuse of buffer handling functions associated with improper string correctness that result in buffer overflows. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | The SQL injection standard highlights the importance of using parameterized queries to mitigate the dangers of SQL injections. By utilizing parameter placeholder values in SQL queries, user input is separated from queries and not treated as SQL code. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, the username is directly concatenated into the SQL query string. This can lead to SQL injections as malicious input can be used in the SQL query. |
| // Noncompliant example  void database\_query() {  // Get username  std::string username = get\_username();  // Concatenates username directly into query string  std::string query = “SELECT \* FROM users WHERE username=’” + username + “’”;  // Executes SQL query, passing the query  execute\_query(query);  } |

| **Compliant Code** |
| --- |
| In the compliant code example, a parameterized SQL query approach is used with a placeholder value in the SQL query and the username is passed as a parameter. This effectively separates SQL logic from user input and reduces the risk of SQL injections. |
| // Compliant example  void database\_query() {  // Get username  std::string username = get\_username();  // Uses parameterized SQL query with a placeholder value ‘?’  std::string query = “SELECT \* FROM users WHERE username=?”;  // Executes SQL query, passing the query and username  execute\_query(query, username);  } |

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

| **Principles(s):** **1, 8.** The SQL Injection Standard is supported by Principle 1: Validate Input Data and Principle 8: Practice Defense in Depth. Adhering to input validation practices through principle 1 will protect against SQL Injection attacks by enforcing the use of parameterized queries and separating user input from SQL queries. Additionally, following Principle 8 with practicing defense in depth can help to reinforce protection from SQL injections by integrating multiple layers of defense focused on separating SQL queries from user input to ensure that user input is not treated as SQL code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0.0git | [alpha.security.taint.TaintPropagation](https://clang.llvm.org/docs/analyzer/checkers.html#alpha-security-taint-taintpropagation-c-c) | The clang-tidy alpha.security.taint.TaintPropagation checker can identify vulnerabilities associated with SQL injection through improper user input validation and sanitation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | The memory protection standard implements proper memory management practices to negate memory vulnerabilities like buffer overflows and memory leaks. Using alternatives like vectors with bounds checking can help reduce these risks and enhance the overall security of the system. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, memory is allocated using an array of size 10. An out of bounds access is then performed that is beyond the range of the array. This can lead to many security issues such as buffer overflows and undefined behavior. |
| // Noncompliant example  void memory\_access() {  // Initialize array  int\* my\_array = new int[10];    // Attempts to access memory out of bounds  my\_array[10] = 5;  // Deallocate memory to avoid memory leaks  delete[] my\_array;  }  int main() {  memory\_access();  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, a vector is used instead of an array as it automatically manages memory and bounds checking. This solution ensures safe memory access within the allocated bounds and mitigates the risks of memory related vulnerabilities. |
| // Compliant example  void memory\_access() {  // Initialize vector  std::vector<int> my\_vector(10);  // Accesses the 9th element of the vector which is in bounds  my\_vector[9] = 5;  }  int main() {  memory\_access();  return 0;  } |

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

| **Principles(s):** **4, 10.** The Memory Protection Standard is supported by Principle 4: Keep It Simple and Principle 10: Adopt a Secure Coding Standard. The keep it simple principle supports the memory protection standard as it reinforces the idea of avoiding code complexity as seen in the noncompliant code example that utilizes arrays instead of vectors for memory management and bounds checking. This makes memory management more difficult and susceptible to vulnerabilities. Also, the adopt a secure coding standard principle can further support the memory protection standard by adopting coding standards that mitigate against memory related vulnerabilities like buffer overflows and memory leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”arrayIndexOutOfBounds”  [CWE: 788](https://cwe.mitre.org/data/definitions/788.html) | Cppcheck has a checker that flags errors of accessing an array index that is out of bounds with the error id “arrayIndexOutOfBounds.” This can help detect and address issues associated with memory management. |
| Clang-Tidy | 19.0.0git | [cppcoreguidelines-pro-bounds-constant-array-index](https://clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines/pro-bounds-constant-array-index.html) | The clang-tidy checker cppcoreguidelines-pro-bounds-constant-array-index also checks for accessing an array index that is out of bounds that can be used to find issues surrounding memory management. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | The assertions standard utilizes assertions to validate assumptions and detect unexpected behavior during program execution. The practice of using assertions can help find and address bugs during the debugging process that can potentially lead to vulnerabilities and exploits in a system. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, the lack of assertions fails to verify assumptions such as that if the data pointer is not null. This can leave the code susceptible to vulnerabilities and unexpected behavior during execution. |
| // Noncompliant code  void process\_data(int\* data, size\_t size) {    // No assertion to check if data is not null  for (size\_t i = 0; i < size; ++i) {  // Do something  }  }  int main() {  int\* data = nullptr;  size\_t size = 10;  process\_data(data, size);  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, assertions are utilized to verify assumptions such as if the data pointer is null. This helps to find and address potential issues during the debugging process that can cause issues during program execution. |
| // Compliant code  void process\_data(int\* data, size\_t size) {    // Uses assertion to check if data is not null  assert(data != nullptr && “Data pointer is null”);  for (size\_t i = 0; i < size; ++i) {  // Do something  }  }  int main() {  int\* data = nullptr;  size\_t size = 10;  process\_data(data, size);  return 0;  } |

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

| **Principles(s):** **2, 8.** The Assertions Standard is supported by Principle 2: Heed Compiler Warnings and Principle 8: Practice Defense in Depth. By heeding compiler warnings with principle 2, assertions can be utilized to validate assumptions and detect unexpected behavior during program execution in the debugging process. Additionally, with principle 8 and practicing defense in depth, incorporating assertions to validate assumptions during the debugging process acts as an additional layer of defense to mitigate against vulnerabilities and exploits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”nullPointer”  [CWE: 476](https://cwe.mitre.org/data/definitions/476.html) | Utilizing the noncompliant code example, cppcheck detected the null pointer error from failing to properly utilize assertions to validate assumptions. |
| Clang-Tidy | 19.0.0git | [clang-analyzer-core.CallAndMessage](https://clang.llvm.org/extra/clang-tidy/checks/clang-analyzer/core.CallAndMessage.html) | The clang-tidy checker clang-analyzer-core.CallAndMessage can be utilized to scan for improper function calls that should be safeguarded by assertions. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exceptions Handling Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | The exceptions handling standard ensures that exceptions are properly handled. This helps prevent potential resource leaks and other vulnerabilities through improper error handling. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, the catch block catches exceptions without specifying the type. This can lead to improper error handling and cleanup which can cause resource leaks and other vulnerabilities. |
| // Noncompliant example  int main() {  // Attempt division by zero  try {  int value = 2 / 0;  std::cout << “Value = “ << value << std::endl;  }  catch (…) {  // Catch that doesn’t handle specific exceptions  std::cerr << “Exception caught” << std::endl;  // Results in improper handling with no potential cleanup  return 1;  }  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, the catch block properly catches with specifying a type. Also, an additional catch block is added for any other unexpected exceptions. This approach avoids vulnerabilities like resource leaks associated with improper exception handling and cleanup. |
| // Noncompliant example  int main() {  // Attempt division by zero  try {  int value = 2 / 0;  std::cout << “Value = “ << value << std::endl;  }  catch (const std:: exception& exception) {  // Catch handling specific exception  std::cerr << “Exception: ” << exception.what() << std::endl;  // Proper handling with cleanup if necessary  return 1;  }  catch (…) {  // Catch handling all additional exceptions  std::cerr << “Exception caught” << std::endl;  return 1;  }  return 0;  } |

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

| **Principles(s):** **9, 8.** The Exceptions Handling Standard is supported by Principle 9: Use Effective Quality Assurance Techniques and Principle 8: Practice Defense in Depth. Through principle 9 and using effective quality assurance techniques, emphasis will be applied in ensuring the proper handling of exceptions to prevent potential resource leaks and other vulnerabilities. Also, utilizing proper exception handling practices contributes to practicing defense in depth as it adds an additional layer of defense in mitigating against vulnerabilities associated with improper exception handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”uninitvar”  [CWE: 457](https://cwe.mitre.org/data/definitions/457.html) | Cppcheck provides the error id “uninivar” for catching exceptions with a catch-all instead of specifying the exception type. This can be useful in identifying issues with exception handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Integer Operations Standard** |
| --- | --- | --- |
| **Integer Operations** | STD-008-CPP | The integer operations standard uses secure integer operation practices to reduce the risks of vulnerabilities. Following the integer operations standard means implementing checks for common integer operations to prevent integer overflow and division by zero. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, checks are not properly implemented for integer overflows or potential division by zero which leaves room for vulnerabilities to be exploited. |
| // Noncompliant code  int addition\_operation(int a, int b) {  // Potential integer overflow  return a + b;  }  double division\_operation(double a, double b) {  // Potential division by zero  return a / b;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, checks are implemented for integer overflows and division by zero with errors properly thrown if detected. This mitigates the potential vulnerabilities associated with these issues. |
| // Compliant code  int addition\_operation(int a, int b) {  // Check first for potential integer overflow  if (a > INT\_MAX – b) {  // Throw error  throw std::overflow\_error(“Detected integer overflow”);  }  return a + b;  }  double division\_operation(double a, double b) {  // Check first for divide by zero  if (b == 0.0) {  // Throw error  throw std::invalid\_argument(“Detected division by zero”);  }  return a / b;  } |

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

| **Principles(s):** **1, 8.** The Integer Operations Standard is supported by Principle 1: Validate Input Data and Principle 8: Practice Defense in Depth. Utilizing input validation with principle 1 will mitigate common integer operation vulnerabilities like integer overflow and division by zero. Principle 8 and practicing defense in depth also contributes by adding these secure integer operation coding practices as another layer of defense, ensuring that checks are implemented for common integer operation errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Visual Studio 2022 | 17.9.0 | [C2124](https://learn.microsoft.com/en-us/cpp/error-messages/compiler-errors-1/compiler-error-c2124?view=msvc-170&f1url=%3FappId%3DDev17IDEF1%26l%3DEN-US%26k%3Dk(C2124)%26rd%3Dtrue) | Visual Studio compiler warning that warns of attempted division by zero, useful for identifying unsafe integer operations. |
| Visual Studio 2022 | 17.9.0 | [C4307](https://learn.microsoft.com/en-us/cpp/error-messages/compiler-errors-1/compiler-error-c2124?view=msvc-170&f1url=%3FappId%3DDev17IDEF1%26l%3DEN-US%26k%3Dk(C2124)%26rd%3Dtrue) | Visual Studio compiler warning that warns of an expression that results in an integer overflowing that can be used in discerning unsafe integer operations. |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”zerodiv”  [CWE: 369](https://cwe.mitre.org/data/definitions/369.html)  error id=”integerOverflow”  [CWE: 190](https://cwe.mitre.org/data/definitions/190.html) | Additionally, cppcheck provides errors for dividing by zero and integer overflows. The error id “zerodiv” indicates detection of dividing by zero and “integerOverflow” detects integer overflow. These checks can help detect issues with integer operations. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Appropriate Exceptions Standard** |
| --- | --- | --- |
| **Appropriate Exceptions** | STD-007-CPP | The appropriate exceptions standard promotes the appropriate use of exceptions for handling exceptional circumstances rather than regular control flow. This helps maintain code clarity, promote maintainability, and prevent potential vulnerabilities within the code. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, exceptions are incorrectly used for regular control flow. This results in unnecessary code complexity that can make it difficult to understand the intended program behavior and prevent potential issues that lead to vulnerabilities. |
| // Noncompliant example  void process\_value(int value) {  try {  // Regular control flow code that shouldn’t throw exceptions  if (value < 0) {  throw std::runtime\_error(“Value is negative”);  }  }  catch (std::exception& exception) {  // Handle Exception  std::cerr << “Exception: “ << exception.what() << std::endl;  }  }  int main() {  int value = -2;  process\_value(value);  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code example, exceptions are only used for exceptional conditions. This reduces code complexity while also promoting readability and maintainability. |
| // Compliant example  void process\_value(int value) {  // Throws exception for exceptional condition  if (value < 0) {  throw std::runtime\_error(“Value is negative”);  }  int main() {  int value = -2;  try {  process\_value(value);  }  catch (std::exception& exception) {  // Handle exception  std::cerr << “Exception: “ << exception.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** **8, 10.** The Appropriate Exceptions Standard is supported by Principle 8: Practice Defense in Depth and Principle 10: Adopt a Secure Coding Standard. By creating a distinct separation for appropriate use of exceptions, the principle for practicing defense in depth is supported as exceptions for truly exceptional behavior will be reserved. This will allow defensive layers that rely on exceptions to better detect potential vulnerabilities. Additionally, the practice of appropriate use of exceptions contributes to the principle adopting a secure coding standard as it promotes the use of a coding standard that follows appropriate exception handling practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 19.0.0git | [bugprone-exception-escape](https://clang.llvm.org/extra/clang-tidy/checks/bugprone/exception-escape.html) | The clang-tidy checker bugprone-exception-escape can be utilized to detect thrown and caught exceptions in areas of the code where they shouldn’t. This requires setting up the functions that expect thrown and caught exceptions to identify the areas in the code that contain inappropriate use of exceptions. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **File Handling Standard** |
| --- | --- | --- |
| **File Handling** | STD-010-CPP | The file handling standard utilizes secure file handling practices to mitigate file security risks. This includes file related vulnerabilities like unauthorized access and resource leaks. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example does not explicitly close the file which opens the possibility of the file not properly closing during execution. This can lead to potential resource leaks and vulnerabilities. |
| // Noncompliant example  int main() {  std::ofstream file(“file.txt”);  file << “File data”;    // File may not be explicitly closed  file.close();  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant code example utilizes a scoped block that automatically closes the opened file when ofstream goes out of scope, preventing potential security risks or resource leaks. This approach also uses error handling in case file operation fails and communicates an error message to the user. |
| // Compliant example  int main() {  // File is open only in scope of block  {  std::ofstream file(“file.txt”);  if (file.is\_open()) {  file << “File data”;  }  else {  std::cerr << “Unable to open file” << std::endl;  // Error handling in the event of file operation failure  return 1;  }  // File automatically closes  }  return 0;  } |

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

| **Principles(s):** **8, 9.** The File Handling Standard is supported by Principle 8: Practice Defense in Depth and Principle 9: Use Effective Quality Assurance Techniques. The principle for practicing defense in depth directly supports this standard by reinforcing the use of defense practices for secure file handling that mitigate file related vulnerabilities. Also, implementing secure file handling practices like using a scoped block for opening files and incorporating error handling for file reading failure support the principle for using effective quality assurance techniques. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | **Settings:**  cppcheck --enable=all --suppress=missingIncludeSystem --inline-suppr --std=c++17 --check-config  **Check flag:**  error id=”resourceLeak”  [CWE: 402](https://cwe.mitre.org/data/definitions/402.html) | Cppcheck provides a checker that can detect resource leaks associated with improper file handling. The error id provided by cppcheck is “resourceLeak” and can be useful in scenarios like the noncompliant code example with resource leaks due to open files because they are not explicitly closed. |

### 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.

Modifying the existing Green Pace DevOps to a DevSecOps process by incorporating automation for the described standards in this policy will first require adjustments to the assess and plan stage in pre-production. Here, appropriate training can be held to utilize the listed automation tools and how they are configured in detecting violations of the coding standards. This applies to the configurations used for Microsoft Visual Studio Code Analysis and compiler warnings, Cppcheck, and clang-tidy checkers. Next, in the verify and test stage during pre-production, the described automation tools can be utilized for thorough vulnerability scanning and functionality, compliancy and security testing.

Moving on to the transition and health check stage in production, the automation tool settings used during the verify and test stage to enforce coding standards can be utilized to configure and deploy security settings for production. Next, during the monitor and detect stage of production, the automation tools can be used to perform static analysis and continuously analyze the code base for security issues and vulnerabilities. Lastly, the automation tools described can be applied to the maintain and stabilize stage of production for any security issues or vulnerabilities addressed during the monitor and detect stage. These tools can be used to ensure that the fixes applied for the found vulnerabilities return the system to baseline while aligning with the coding standards of this Green Pace security policy.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Possible | Low | Medium | 2 |
| STD-003-CPP | Medium | Possible | Low | Medium | 2 |
| STD-004-CPP | High | Possible | High | High | 4 |
| STD-005-CPP | High | Likely | High | High | 5 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | Medium | Possible | Medium | Medium | 3 |
| STD-008-CPP | High | Possible | High | High | 4 |
| STD-009-CPP | High | Possible | High | High | 4 |
| STD-010-CPP | Medium | Possible | Medium | Medium | 3 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the encryption of data that is stored in a resting state. This can be data that is stored in a physical hard drive, database, or even in the cloud. Enforcement of this policy will allow for the security of data at rest by encrypting the stored data so that, in the event of a compromise, the data remains secure. Applying encryption at rest can be accomplished through full disk encryption, database encryption, and cloud assets encryption.  Source: <https://cyscale.com/blog/types-of-encryption/> |
| Encryption in flight | Encryption in flight is the encryption of data that is moving or in transit. This usually refers to the encryption of data that is being sent over a network. By following this policy, data being sent over a network will remain secure by encrypting any sensitive data during transit. Encryption in flight can be applied by using SSL/TLS transport layer protocols to encrypt data in transit.  Source: <https://cyscale.com/blog/types-of-encryption/> |
| Encryption in use | Encryption in use is the encryption of data that is being utilized by an application or system. This refers to the practice of encrypting data that is being processed or used by the application or system. Adhering to this policy ensures that data remains secure by encrypting sensitive data during data processing. A method for applying encryption for data in use is utilizing a memory encryption tool such as SEV that encrypts RAM for data in use.  Source: <https://cyscale.com/blog/types-of-encryption/> |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the process of verifying and identifying those trying to access the application or system. This process checks credentials for access, commonly implemented as a username and password system but can also check for physical keys or biometric keys like a fingerprint. Following the authentication policy ensures that appropriate user login and password requirements are enforced for all users, old and new, so that only those with the proper credentials are granted access to data or services that the application or system provides.  Source: <https://www.fortinet.com/resources/cyberglossary/aaa-security> |
| Authorization | Authorization is the process of granting system or application privileges to a user. With authorization, the user level of access is defined so that only the necessary permissions are granted for the expected uses of the application or system. Enforcing this policy includes adopting a principle of least privilege approach and ensuring that every user within the system is only granted the necessary privilege in completing their expected tasks. This can help mitigate the risk of unauthorized users compromising sensitive data in the application or system.  Source: <https://www.fortinet.com/resources/cyberglossary/aaa-security> |
| Accounting | Accounting refers to the process of logging and keeping track of user activity and data within a system. Accounting covers the various logging of activities and can be changes to the database, addition of new users, as well as the files accessed by users. Adhering to this policy means keeping thorough system and user logs that can be utilized to detect suspicious user or system activity.  Source: <https://www.fortinet.com/resources/cyberglossary/aaa-security> |

**\***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 logsThe 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 | 03/31/2024 | Added Milestone Three requirements | Michael Schultz |  |
| 1.2 | 4/20/2024 | Added Project One requirements | Michael Schultz |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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