

# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc159862919)

[Contents 1](#_Toc159862920)

[Overview 3](#_Toc159862921)

[Purpose 3](#_Toc159862922)

[Scope 3](#_Toc159862923)

[Module Three Milestone 3](#_Toc159862924)

[Ten Core Security Principles 3](#_Toc159862925)

[C/C++ Ten Coding Standards 4](#_Toc159862926)

[Data Type Coding Standard 5](#_Toc159862927)

[Data Value Coding Standard 7](#_Toc159862928)

[String Correctness Coding Standard 9](#_Toc159862929)

[SQL Injection Coding Standard 11](#_Toc159862930)

[Memory Protection Coding Standard 13](#_Toc159862931)

[Assertions Coding Standard 15](#_Toc159862932)

[Exceptions Coding Standard 17](#_Toc159862933)

[Deserialization Coding Standard 19](#_Toc159862934)

[External Libraries Coding Standard 21](#_Toc159862935)

[Privileges Coding Standard 22](#_Toc159862936)

[Defense-in-Depth Illustration 24](#_Toc159862937)

[Project One 24](#_Toc159862938)

[Revise the C/C++ Standards 24](#_Toc159862939)

[Risk Assessment 24](#_Toc159862940)

[Automated Detection 24](#_Toc159862941)

[Automation 24](#_Toc159862942)

[Summary of Risk Assessments 25](#_Toc159862943)

[Create Policies for Encryption and Triple A 25](#_Toc159862944)

[Map the Principles 27](#_Toc159862945)

[Audit Controls and Management 28](#_Toc159862946)

[Enforcement 28](#_Toc159862947)

[Exceptions Process 28](#_Toc159862948)

[Distribution 29](#_Toc159862949)

[Policy Change Control 29](#_Toc159862950)

[Policy Version History 29](#_Toc159862951)

[Appendix A Lookups 29](#_Toc159862952)

[Approved C/C++ Language Acronyms 29](#_Toc159862953)

[References 30](#_Toc159862954)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input data must always be verified. This is the first line of defense against malicious external actors. Simply checking the input data type, length, and format (*Secure product design cheat sheet*) will mitigate a large portion of attempted cross-site scripting, injection attacks, and buffer overflows (Morrow, 2023). It is the standard for Green Pace to be wary of all external data sources. |
| 1. Heed Compiler Warnings | Compiler warnings are a built-in indicator of potential vulnerabilities. Any compiler warnings are to be amended as soon as possible to avoid building on top of vulnerable code (Morrow, 2023). The Green Pace standard is to turn over every rock by using the highest warning level available when compiling code (Seacord, 2018). |
| 1. Architect and Design for Security Policies | The best defense starts at the foundational level. The Green Pace standard includes having a security mindset from the planning phase through the maintenance phase of every project. A project built on software design and architecture with integrated security policies is a must (Seacord, 2018). |
| 1. Keep It Simple | At Green Pace, simplicity is key. Coding with a mindset of “simple and clean” leads to far less bugs that are also easier to find (Seacord, 2018). Our principle of simplicity leads to better results for the quality assurance team too, since they can achieve higher test coverage of the existing code. |
| 1. Default Deny | Green Pace employs the policy to deny by default. Users are denied access to all system functionalities until their individual authorization is proven (Morrow, 2023). Instead of revoking certain system privileges on a user basis, specific privileges are granted on a user basis (Seacord, 2018). |
| 1. Adhere to the Principle of Least Privilege | Tasks assigned to Green Pace developers must only be done so with the lowest privileges required to complete them (Seacord, 2018). To prevent leaking information that can inform writers of injection attacks, code is protected on a need-to-know basis (Morrow, 2023). Users assigned tasks that require elevated permissions will have the additional permissions revoked upon completion of said tasks (Seacord, 2018). |
| 1. Sanitize Data Sent to Other Systems | Any Green Pace sent to complex subsystems must first be sanitized to match the security-related requirements of that subsystem (Seacord, 2015). Formatting that does not conform to the subsystem on the other side of a trust boundary can be invoked by attackers through command, SQL, or other injection types (Seacord, 2018). |
| 1. Practice Defense in Depth | Green Pace employs a defense in depth security system. Our multiple layers of defense ensure the best odds of mitigating malicious attacks (Seacord, 2018). Any layers of defense that prove to be inadequate must be meticulously analyzed and reconfigured. |
| 1. Use Effective Quality Assurance Techniques | Green Pace’s quality assurance policy includes many techniques. We commission external penetration testing by third parties who generate an exhaustive report of vulnerabilities to be addressed (Seacord, 2018). Internal testing by Green Pace cybersecurity specialists includes fuzz testing and source code audits (Seacord, 2018). |
| 1. Adopt a Secure Coding Standard | Green Pace applies these ten principles of security as our secure coding standard. This standard is to be applied to all development platforms and languages used at Green Pace (Seacord, 2018). For additional information, please refer to official Open Web Application Security Project (OWASP) resources. |

### 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.

#### Data Type Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | **Eliminate variadic functions**. Variadic functions are those that accept multiple arguments of various data types (Britton, 2023). This is an insecure format since data type checking is an important barrier to an attacker attempting to input malicious scripts. |

| **Noncompliant Code** |
| --- |
| The following is an example of a simple variadic function. This function accepts parameters of any data type and attempts to return their sum. Data type checking is unproductive here since data of any type is accepted. |
| template<typename T, typename… Types>  T sum(T arg1, Types… args) {  return arg1 + sum(args…);  }  template<typename T>  T sum(T lastArg) {  return lastArg;  } |

| **Compliant Code** |
| --- |
| Use a format that is specific about the data types and number of variables accepted. Use function overloading to accept parameters of various data types and numbers of variables. Data type checking is possible with this format. |
| int sum(int a, int b, int c) {  return a + b + c;  }  float sum(float a, float b, float c) {  return a + b + c;  }  ... |

| **Principles(s):** *Validate Input Data*: Eliminating the use of variadic functions simplifies the process of validating all input data by first ensuring only input of the expected data type is accepted. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Less likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 7.6.0.83110 | [User-defined types should not be passed as variadic arguments](https://rules.sonarsource.com/cpp/RSPEC-5270/?search=variadic) | Use in conjunction with the second checker to ensure only the exact number of variables with the expected data type(s) are accepted. |
| Sonar | 7.6.0.83110 | [Functions should not be defined with a variable number of arguments](https://rules.sonarsource.com/cpp/RSPEC-923/?search=variadic) | Use in conjunction with the first checker to ensure only the exact number of variables with the expected data type(s) are accepted. |

#### Data Value Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Check all data values**. Data values need to be checked to ensure functions behave as expected. For example, inputs of type int can have boundaries placed on them where appropriate. Entering a person’s age would not accept a negative number or a positive number greater than around 120 years. Checking data values can also contribute to catching and preventing malicious acts like numeric overflow and underflow. This act can allow attackers to gain access to memory locations outside of this scope that they can subsequently manipulate. |

| **Noncompliant Code** |
| --- |
| The following code assumes that all parameters are positive numbers. The code does not check whether each subsequent increment to the result variable will overflow or underflow it. See the compliant code block to understand how to prevent numeric overflow or underflow. |
| int add\_numbers(int start, int increment, int steps)  {  int result = start;  for (int i = 0; i < steps; ++i) {  result += increment;  }  return result;  } |

| **Compliant Code** |
| --- |
| The following code assumes that all parameters are positive numbers. The result variable’s value is being checked for potential numeric overflow. If the current increment would exceed the maximum value of the int data type, the code does not perform that increment and instead displays an error message. Checks like these are important for additional data types as well. |
| int add\_numbers(int start, int increment, int steps)  {  int result = start;  for (int i = 0; i < steps; ++i) {  if ((INT\_MAX – increment < result) || (result + increment) < 0) {  std::cout << “Increment prevented: Value out of bounds.” << endl;  }  result += increment;  }  return result;  } |

| **Principles(s):** *Validate Input Data*, *Heed Compiler Warnings*: The validation of input data also includes checking the value itself, even when it is of the expected data type. The compiler may catch these, but do not depend on it to do so. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 7.6.0.83110 | [Integral operations should not overflow](https://rules.sonarsource.com/cpp/RSPEC-3949/?search=overflow) | Prevent integer wrapping before it happens. |
| Sonar | 7.6.0.83110 | [Zero should not be a possible denominator](https://rules.sonarsource.com/cpp/RSPEC-3518/?search=overflow) | To prevent a specific type of numeric overflow. |

#### String Correctness Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | **Check all strings for correctness**. Many malicious attacks, such as purposeful buffer overflows, can be prevented when all strings are checked for correctness. For example, strings that are accepted as input in the form of character arrays must have their length checked to prevent an attacker from gaining access to memory outside of the allocated input space. |

| **Noncompliant Code** |
| --- |
| This code accepts user input into a character array of size 20, but the field to input text does not have a character limit. Therefore, this code can cause a buffer overflow that a malicious attacker can exploit to gain access to other memory locations. |
| const std::string account\_number = “CharlieBrown42”;  char user\_input[20];  std::cout << “Enter a value: ”;  std::cin >> user\_input;  std::cout << “You entered: ” << user\_input << std::endl;  std::cout << “Account Number = ” << account\_number << std::endl; |

| **Compliant Code** |
| --- |
| The above code can be modified as follows to prevent buffer overflows. Create a variable of type const int to hold the maximum character count allowed. Then, use the C++ standard library method setw() to set the input stream’s width parameter to the custom maximum character count. In this example, only the first 20 input characters will be stored in user\_input. |
| const int INPUT\_MAX = 20;  const std::string account\_number = “CharlieBrown42”;  char user\_input[20];  std::cout << “Enter a value: ”;  std::cin >> std::setw(INPUT\_MAX) >> user\_input;  std::cout << “You entered: ” << user\_input << std::endl;  std::cout << “Account Number = ” << account\_number << std::endl; |

| **Principles(s):** *Validate Input Data*, *Heed Compiler Warnings*: The validation of input data also includes checking the string value itself, even when a string is expected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 7.6.0.83110 | [Memory access should be explicitly bounded to prevent buffer overflows](https://rules.sonarsource.com/cpp/RSPEC-3519/?search=string%20length) | This is one way to prevent obfuscated memory from being exposed. |
| Sonar | 7.6.0.83110 | [Using hardcoded IP addresses is security-sensitive](https://rules.sonarsource.com/cpp/RSPEC-1313/?search=validate%20string) | Checking string correctness also includes making sure sensitive information is not hardcoded. |

#### SQL Injection Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **Check for potential SQL injection attacks in all relevant locations**. Code that is vulnerable to SQL injection can be exploited by attackers to gain access to sensitive database content. All user input fields must be checked for potential SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| The following code is an example of a query to an SQL database. Here, the query is not checked for SQL injections and just returns true. An attacker can potentially access sensitive database content. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  return true;  } |

| **Compliant Code** |
| --- |
| A secure version of this method checks for potential SQL injections. This example is checking for SQL injection attacks of the format “OR value=value” which can cause an SQL IF statement to evaluate as true and grant the attacker access to sensitive database content. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  size\_t or\_statement = sql.find(“ or ”);  if (or\_statement != std::string::npos)  {  std::string after\_or = sql.substr(or\_statement + 4);  after\_or.erase(after\_or.begin(), std::find\_if(after\_or.begin(), after\_or.end(), [](unsigned char ch) {  return !std::isspace(ch);  }));  if (after\_or.find(‘=’) != std::string::npos)  {  std::cout << “Potential SQL injection attack blocked.” << endl;  return false;  }  return true;  }  } |

| **Principles(s):** *Validate Input Data*, *Default Deny*: Sensitive database content should be well protected. There should be active monitoring for SQL injection attempts, and all system functionalities should be denied until authorization is proven. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 7.6.0.83110 | [Format strings should be used correctly](https://rules.sonarsource.com/cpp/RSPEC-3457/?search=sql%20string) | There are many ways that malicious actors can craft their input to manipulate source code. This is one method to prevent. |
| Sonar | 7.6.0.83110 | [String literals should not be concatenated implicitly](https://rules.sonarsource.com/cpp/RSPEC-3728/?search=sql%20string) | This is another method that could be used to circumvent the detection of suspicious input. |

#### Memory Protection Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **All pointers must be smart pointers**. Memory protection metrics can be significantly improved when only smart pointers are used. These exception-safe alternatives help prevent resource and memory leaks (*Smart pointers (modern C++)* 2021). Smart pointers include those of the type auto\_ptr, unique\_ptr, shared\_ptr, and weak\_ptr (*Smart pointers in C++* 2024). |

| **Noncompliant Code** |
| --- |
| This code block demonstrates the use of a raw pointer which will likely cause a memory leak or other type of vulnerability. |
| void myPointer() {  Dog\* myDog = new Dog(“Shetland Sheepdog”, “Chip”);  if (myDog) {  std::cout << “Dog breed: ” << myDog->getBreed() << std::endl;  std::cout << “Name: ” << myDog->getName() << std::endl;  }  delete myDog;  } |

| **Compliant Code** |
| --- |
| The adjusted code block employs a smart pointer which has built-in memory management. |
| void myPointer() {  unique\_ptr<Dog> myDog(new Dog(“Shetland Sheepdog”, “Chip”));  if (myDog) {  std::cout << “Dog breed: ” << myDog->getBreed() << std::endl;  std::cout << “Name: ” << myDog->getName() << std::endl;  }  } // myDog is automatically deleted here |

| **Principles(s):** *Adopt a Secure Coding Standard*: Green Pace uses C++ which includes the ability to manipulate pointers and manually manage memory. With this added functionality comes the responsibility of strict compliance with our memory protection policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Less likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 7.6.0.83110 | [Dynamically allocated memory should be released](https://rules.sonarsource.com/cpp/RSPEC-3584/?search=memory) | The freedom to use pointers comes with the responsibility of strict memory management. |
| Sonar | 7.6.0.83110 | [Memory should not be managed manually](https://rules.sonarsource.com/cpp/RSPEC-5025/?search=memory) | To ensure proper memory management, the developer can set up tools to do it automatically. |
| Sonar | 7.6.0.83110 | [Freed memory should not be used](https://rules.sonarsource.com/cpp/RSPEC-3529/?search=memory) | Accessing memory that has been freed can lead to undefined behavior. |

#### Assertions Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Assertion conditions must be static (i.e., constant expressions)** (Choudhary). Assertion expressions that are nonstatic will result in expressions that evaluate to different values in the release version versus the debug version (where the assertion is employed) (*C/C++ assertions* 2023). This can cause unexpected behavior in the release version, thus giving attackers a foothold to reach code meant to be obfuscated. |

| **Noncompliant Code** |
| --- |
| This code features a nonstatic assertion condition in which the variable x is being decremented before its value is checked if it is greater than zero. This expression will evaluate to a different value in the release version. This decrement does not occur in the release version and is meant for the debug version where assertions are relevant. |
| void myFunction(int x) {  assert(x-- > 0);  ...  } |

| **Compliant Code** |
| --- |
| The adjusted code block features a static assertion condition that will evaluate to the same value regardless of if the code is in debug mode or release mode, leading to far more predictable code. |
| void myFunction(int x) {  assert(x > 0);  ...  } |

| **Principles(s):** *Use Effective Quality Assurance Techniques*, *Adopt a Secure Coding Standard***:** Assertions are a great way to test the quality of code. However, unexpected behavior caused by assertions can be a gold mine for malicious actors. Stay consistent with assertions by only including static expressions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Multiple (see link) | Multiple (see link) | [Various checkers available](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression). E.g., Clang version 3.9 provides the misc-static-assert checker | Various checks for C-type assertions. |

#### Exceptions Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | **Always use exceptions where errors may occur** (*Modern C++ best practices for exceptions and error handling* 2022). The exception format in C++ forces code to recognize errors and to handle them appropriately right then and there, thus preventing faulty code from being exposed. When used in conjunction with smart pointers, exceptions are a great way to ensure resource cleanup (*Modern C++ best practices for exceptions and error handling* 2022). |

| **Noncompliant Code** |
| --- |
| In the code below, some invalid input is passed as an argument to the function myFunction(). This code does not include any method of telling the program what to do when this happens. Thus, the invalid input can cause unexpected behavior such as a numeric overflow or underflow, a buffer overflow, etc. It can also cause a memory leak since resource cleanup was not considered here. |
| myFunction(“[Invalid input]”); |

| **Compliant Code** |
| --- |
| The code is adjusted here to comply with exception formatting. Attempting to call myFunction() with invalid input causes the code to throw an exception that is then caught. Finally, an error message is displayed. |
| try {  myFunction(“[Invalid input]”);  }  catch (invalid\_argument& e) {  cerr << e.what() << endl;  return -1;  } |

| **Principles(s):** *Use Effective Quality Assurance Techniques*, *Adopt a Secure Coding Standard*: Exceptions are another great tool to check the quality of code. Remember to remove them in release versions to prevent exposing source code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| N/A | N/A | N/A | An automated tool is not necessary since exceptions are very distinct depending on the project. Always use them to catch potential unexpected behaviors. |

#### Deserialization Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Deserialization** | STD-008-CPP | **Do not use the deserialization functionalities native to C++** . Deserialization is the process of taking structured data, like a JSON object or XML document, and reconstructing it as an object (*Deserialization cheat sheet*). Deserialization functionalities that are native to a programming language can be manipulated by a malicious actor, granting them the ability to employ attacks like denial-of-service (DoS) or even remote code execution (RCE) (*Deserialization cheat sheet*). |

| **Noncompliant Code** |
| --- |
| This code block overloads the >> operator to accept a stream’s object data and use that data to reconstruct it as an object. This code builds an object from any input data without discretion. |
| friend std::istream& operator>>(std::istream& is, MyClass& obj) {  is >> obj.data;  return is;  } |

| **Compliant Code** |
| --- |
| The adjusted code first checks the result of the input stream. If an error was encountered due to, for example, the input being invalid, then an exception is thrown and the object is not built from the invalid data. |
| friend std::istream& operator>>(std::istream& is, MyClass& obj) {  if (!(is >> obj.data)) {  throw std::invalid\_argument(“This deserialized data does match the expected criteria.”);  }  return is;  } |

| **Principles(s):** *Validate Input Data*, *Default Deny*, *Sanitize Data Sent to Other Systems*: Sanitizing our data before transmission can help mitigate exposure of our source code. All input must be validated, and if it resembles JSON or other structured data, it will be flagged for further inspection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Less likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| N/A | N/A | N/A | See [this resource](https://learn.snyk.io/lesson/insecure-deserialization/) for some further information about insecure deserialization. In general, all input should be checked for structured data. If structured data is allowed, check for legitimacy. |

#### External Libraries Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **External Libraries** | STD-009-CPP | **Do not use any untrusted external libraries**. Only reputable, trusted external libraries accessed from official links may be used (Madou, 2020). The use of untrusted libraries is a large source of code vulnerabilities. Be especially cautious when using Docker, since code can develop a permanent dependency on an external library the moment it is embedded into a Docker image (Madou, 2020). |

| **Noncompliant Code** |
| --- |
| The difference between noncompliant code and compliant code for this example is just an include command. However, the real difference is made by the developer’s detailed research. This external library was picked by a developer who wanted a shortcut to finish their project. It is a random library by an unknown author. |
| #include “suspiciouslibrary.h” |

| **Compliant Code** |
| --- |
| The external library included in the compliant code has a great reputation and is used by many esteemed organizations. The code is open source and deemed safe when used as intended. |
| #include “vettedlibrary.h” |

| **Principles(s):** *Use Effective Quality Assurance Techniques*: All external libraries must pass an extensive Green Pace security analysis before use. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Sonar | 7.6.0.83110 | [Code annotated as deprecated should not be used](https://rules.sonarsource.com/cpp/RSPEC-1874/?search=api) | Implemented APIs with deprecated code are a significant threat to security. All APIs should be vetted for legitimacy before implementation. |

#### Privileges Coding Standard

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Privileges** | STD-010-CPP | **Code with the principle of least privilege (PoLP) in mind** (*Increase application security with the principle of least privilege* 2023). Write code that is adaptable to varying levels of user access to data and operations. For security purposes, each individual user is granted only the lowest privileges required to get their job done (*Increase application security with the principle of least privilege* 2023). |

| **Noncompliant Code** |
| --- |
| This code grants all users permission to write to any file without discretion. |
| void writeToMyFile(const std::string& data, const std::string& myFile) {  std::ofstream fileStream(myFile, std::ios::out);  if (fileStream.is\_open()) {  fileStream << data;  std::cout << “Successful data write.\n”;  }  else {  std::cerr << “Error opening file for writing.\n”;  }  fileStream.close();  } |

| **Compliant Code** |
| --- |
| This code has an integrated privileges check. Only users who have the necessary permissions to do so may write to certain files. |
| bool checkUserPrivileges() {  // Code that checks the current users’ privileges  // If the privileges are sufficient, return true  return true;  }  void writeToMyFile(const std::string& data, const std::string& myFile) {  if (!checkUserPrivileges()) {  std::cerr << “You do not have permission to write to this file.\n”;  return;  }  std::ofstream fileStream(myFile, std::ios::out);  if (fileStream.is\_open()) {  fileStream << data;  std::cout << “Successful data write.\n”;  }  else {  std::cerr << “Error opening file for writing.\n”;  }  fileStream.close();  } |

| **Principles(s):** *Default Deny*, *Adhere to the Principle of Least Privilege*: System users are given authorization for just enough privileges to complete their respective tasks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Less likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| N/A | N/A | N/A | While this is a system administration requirement, written code should accommodate all user groups with various levels of permissions. |

### 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.

To successfully implement the new Green Pace DevSecOps infrastructure, the automation that checks for compliance with our strict coding policies is planned for during every project’s design phase. Any custom rules are written in the build phase, and the tests are run during the verify and test phase. This is an iterative process that must be reassessed during every health check and monitoring phase. Any persisting vulnerabilities are flagged. If existing vulnerabilities are high risk, relevant services are turned off until security is restored. If damage has occurred within this iteration, developers will work alongside Green Pace cybersecurity specialists to restore a stable state before proceeding to the next planning phase.

### 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** | Medium | Less likely | Medium | Medium | 3 |
| **STD-002-CPP** | Medium | Less likely | Low | Medium | 2 |
| **STD-003-CPP** | Medium | Less likely | Low | Medium | 2 |
| **STD-004-CPP** | High | Likely | High | High | 4 |
| **STD-005-CPP** | Medium | Less likely | Medium | Medium | 3 |
| **STD-006-CPP** | Medium | Unlikely | Medium | Medium | 2 |
| **STD-007-CPP** | Medium | Unlikely | Medium | Medium | 2 |
| **STD-008-CPP** | High | Less likely | High | High | 5 |
| **STD-009-CPP** | High | Likely | High | High | 5 |
| **STD-010-CPP** | High | Less likely | High | High | 5 |

### 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 | Data at rest is that which is stored (e.g. on a flash drive, hard drive, or in the cloud) or otherwise not interfacing with a third party (*Encryption: Understanding data at rest vs. in transit* 2021). While data at rest is often less vulnerable than data in transit, robust encryption is still critical due to it typically containing the most sensitive company information (*Encryption: Understanding data at rest vs. in transit* 2021). |
| Encryption in flight | Data in flight is that which is transported, perhaps over the internet, over the Green Pace intranet, or to the cloud (*Encryption: Understanding data at rest vs. in transit* 2021). All data in flight must be encrypted, preferably with asymmetric encryption using a strong cipher. |
| Encryption in use | Data in use is that which is being accessed by a Green Pace CPU (*Encryption: Understanding data at rest vs. in transit* 2021). Any data that can be viewed, updated, deleted, or otherwise accessed by the system must be encrypted in a way that will not damage or crash relevant system applications (*Encryption: Understanding data at rest vs. in transit* 2021). |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Proper authentication of all users is vital to security upkeep. Credentials are needed to identify users, and passwords must meet strong requirements. Ideally, users will utilize a password manager that can generate a very strong password. Unauthenticated users are not granted access to any level of the Green Pace system. |
| Authorization | Our authorization levels are meticulously designed to only grant each specific user group enough permissions to complete the tasks they are assigned. The principle of least privilege (PoLP) is used the ensure the privacy of sensitive Green Pace data. For example, only system administrators are granted permission to add new users. |
| Accounting | Accounting is our way of logging system activity (e.g., changes to a system database or files accessed by users) in real time. Full system audits occur regularly where system events are automatically and manually analyzed to detect, flag, and prevent suspicious activity. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 02/03/2024 | 3-2 Milestone Version | Amanda Purnhagen | Ahlam Alhweiti |
| 3.0 | 02/26/2024 | 6-2 Project One Version | Amanda Purnhagen | Pending approval |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

## References

Britton, J. (2023, April 20). *DCL50-CPP. do not define a C-style variadic function*. Confluence. <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function>

*C/C++ assertions*. Microsoft Learn. (2023, March 9). <https://learn.microsoft.com/en-us/visualstudio/debugger/c-cpp-assertions?view=vs-2019#BKMK_Side_effects_of_using_assertions>

Choudhary, V. (n.d.). *Simple and static assertion (assert) in C programming language*. Developer Insider. <https://developerinsider.co/simple-and-static-assertion-assert-in-c-programming-language/>

*Deserialization cheat sheet*. OWASP Cheat Sheet Series. (n.d.). <https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html>

*Encryption: Understanding data at rest vs. in transit*. DataLocker Inc. (2021, September 24). <https://datalocker.com/blog/technology/encryption/encryption-at-rest-vs-in-transit-effectively-encrypt-identifiable-information/>

*Increase application security with the principle of least privilege*. Microsoft Learn. (2023b, October 23). <https://learn.microsoft.com/en-us/entra/identity-platform/secure-least-privileged-access>

Madou, M. (2020, June 15). *Coders conquer security infrastructure as code series - using components from untrusted sources*. Secure Code Warrior. <https://www.securecodewarrior.com/article/coders-conquer-security-infrastructure-as-code-series-using-components-from-untrusted-sources>

*Modern C++ best practices for exceptions and error handling*. Microsoft Learn. (2022, October 16). <https://learn.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=msvc-170&redirectedfrom=MSDN>

Morrow, S. (2023, August 22). *What is secure coding and why is it important?* VPNOverview. <https://vpnoverview.com/internet-safety/business/what-is-secure-coding/#:~:text=Secure%20code%20will%20help%20to,Software%20vulnerabilities%20are%20rampant>

Seacord, R. (2015, April 28). *Input validation and data sanitization*. Confluence. <https://wiki.sei.cmu.edu/confluence/display/java/Input+Validation+and+Data+Sanitization#:~:text=Data%20sanitization%20is%20the%20process,output%20across%20a%20trust%20boundary>

Seacord, R. (2018, May 2). *Top 10 secure coding practices*. Confluence. <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCommentId=88044413>

*Secure product design cheat sheet*. OWASP Cheat Sheet Series. (n.d.). <https://cheatsheetseries.owasp.org/cheatsheets/Secure_Product_Design_Cheat_Sheet.html>

*Smart pointers (modern C++)*. Microsoft Learn. (2021, August 2). <https://learn.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170>

*Smart pointers in C++*. GeeksforGeeks. (2024, January 11). <https://www.geeksforgeeks.org/smart-pointers-cpp/>