**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 | Any time an application takes user input, the input data needs to be validated. This means that it should be tested to ensure that it follows the defined parameter for the input, including matching the correct data type, format, and length for the input. |
| 1. Heed Compiler Warnings | Code should be refactored to compile without warnings from the compiler. Code without errors may compile, but the warnings generated by the compiler point to potentially unsecure code. Compiler warnings should not be ignored. |
| 1. Architect and Design for Security Policies | The security policy needs to be integrated into the design phase. Code needs to be designed with security in mind. Security should be a consideration at every phase of software development, not left until the end. |
| 1. Keep It Simple | Security measures should be kept simple. Overcomplicated measures can actually be less secure than simple measures. If the security measures are difficult to use, it can lead users to circumvent them for convenience. For example, if the system requires frequent password changes, users may opt to keep an unsecured file on their desktop with their current password. |
| 1. Default Deny | Deny by default means that anything that anyone attempting to access the system that has not been permitted should be denied. To follow this principle, a whitelist should be employed to permit access, rather than using a blacklist to block access. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege states that every user should only have permission to access the functions that they require to complete necessary tasks. No user should have permission for a function that is unnecessary for their work with the system. |
| 1. Sanitize Data Sent to Other Systems | Data needs to be sanitized to prevent injection attacks. The method of sanitizing the data depends on the type of system the data is sent to. If the data is sent to an SQL database, prepared statement should be used. Special characters such as quotations, less than, and greater than character should be escaped, as they can be interpreted as the start of commands. |
| 1. Practice Defense in Depth | Multiple layers of defense should be employed. These layers should overlap, preventing attackers from gaining access by bypassing a single layer. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques are an important part of software security. Software should undergo several rounds of testing, including static testing, unit testing, load testing, penetration testing, and code audits. |
| 1. Adopt a Secure Coding Standard | A coding standard should be defined and adopted to ensure that all developers are adhering to the secure coding principles. |

### 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** | **Avoid Integer Division** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | The use of integer division can lead to logical errors and bugs. By using a floating-point data type, these bugs can be avoided. |

| **Noncompliant Code** |
| --- |
| This code will result in the average equaling 5, because anything after the decimal point is truncated. |
| int a = 5;  int b = 6;  int average = (a + b)/2; |

| **Compliant Code** |
| --- |
| This code results in the correct answer of 5.5 for the average. |
| int a = 5;  int b = 6;  double average = (a + b)/2.0; |

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

| **Principles(s):** Heed Compiler Warnings – Some compilers will warn you of integer division that will truncate data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 18.1.8 | -wconversion | Warns when implicit conversion may alter a value |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Check for Numeric Overflow** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Numeric overflow can cause undefined behavior. When there is the possibility of overflow, overflow should be checked for and handled. |

| **Noncompliant Code** |
| --- |
| This block of code will result in numeric overflow when a is assigned with the result of a + b. |
| short a = 17000;  short b = 16000;  a = a + b; |

| **Compliant Code** |
| --- |
| This code checks to see if the operation will result in numeric overflow before performing the operation. If overflow occurs, an exception is thrown that can be handled elsewhere. |
| short a = 17000;  short b = 16000;  if(a > std::numeric\_limits<short>::max() – b){  throw std::exception(“Overflow occurred”);  }  else{  a = a + b;  } |

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

| **Principles(s):** Validate Input Data – When a numeric data type is used for user input, there is the possibility that the user will enter a value that is greater than the data type maximum. User input should be validated to ensure that numeric overflow does not occur. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 12.14 | - signed integer overflow | (only enabled when --platform is used) |
| SonarLint | 8.2.1 | [Integral operations should not overflow](https://rules.sonarsource.com/cpp/RSPEC-3949/) |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Utilize std::string over C-Strings** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The standard library string provides automatic memory management that prevents many of the errors that arise from using c-strings. |

| **Noncompliant Code** |
| --- |
| This code block causes a buffer overflow error, because the defined string only has room for 4 characters in memory. |
| char str[4];  char str[0] = ‘h’;  char str[1] = ‘e’;  char str[2] = ‘l’;  char str[3] = ‘l’;  char str[4] = ‘o’; |

| **Compliant Code** |
| --- |
| When utilizing a standard library string, the memory of the string is dynamically allocated, so buffer overflow will not occur. |
| std::string str;  str = “hello”; |

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

| **Principles(s):** Keep it simple – The use of standard library strings is simpler than the memory management necessary for c-strings. Utilizing them will help reduce errors and make code more secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ClangTidy | 20.0.0 | modernize-use-string |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Utilize Prepared Statement for SQL Queries** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection is possible when the user input is recognized as SQL code. By utilizing prepared statements, user input will not be considered SQL code. |

| **Noncompliant Code** |
| --- |
| If the query in this code is executed, there is a danger of SQL injection. An attacker could add the statement OR 1=1 to their input, and it would be recognized as SQL code and executed. |
| std::string username;  std::string password;  std::cout << “Enter your username\n”;  std::cin >> username;  std:: cout << “Enter your password\n”;  std::cin >> password;  std::string query = “SELECT \* FROM users WHERE username = ‘” + username  + “’ AND password = ‘” + password + “’;”; |

| **Compliant Code** |
| --- |
| By using prepared statements, the code is forced to view the user input as a string. Even if an attacker attempt to inject SQL code, their code will be viewed as a string object. |
| std::string username;  std::string password;  sql::Connection \*con;  sql::PreparedStatement stmt;  std::string query = “SELECT \* FROM users WHERE username = ? AND password  = ?”  std::cout << “Enter your username\n”;  std::cin >> username;  std:: cout << “Enter your password\n”;  std::cin >> password;  stmt = con->prepareStatement(query);  stmt->setString(1, username);  stmt->setString(2, password); |

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

| **Principles(s):** Sanitize Data Sent to Other Systems – The use of prepared statements is a form of sanitizing input that is sent to the database.  Validate User Input - Unvalidated user input can lead to SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQL Map | 1.8 | Full support for six SQL injection techniques: boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries and out-of-band. |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Free Dynamically Allocated Memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Unlike Java, C++ does not have automatic garbage collection. As such, memory that is dynamically allocated should be expressly freed when it is no longer needed. |

| **Noncompliant Code** |
| --- |
| Every time this function is called, it dynamically allocates a new integer. Since this memory is never freed, the memory is reserved while the application is running. If the function is called repeatedly during runtime, there is the possibility that the system will run out of memory. |
| int function(int a){  int \*p = new int;  \*p = 10;  int result = a + \*p;  return result;  } |

| **Compliant Code** |
| --- |
| The update to the function deletes the dynamic pointer and frees the allocated memory when it is no longer needed. This function can be called repeatedly without fear of running out of memory. |
| int function(int a){  int \*p = new int;  \*p = 10;  int result = a + \*p;  delete p;  return result;  } |

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

| **Principles(s):** Architect and Design for Security Policies – Whenever the design for the system requires dynamic memory allocation, you should also plan for deallocation.  Heed Compiler Warnings – Some compilers will warn you of dynamically allocated memory that is never freed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | 8.2.1 | [Dynamically allocated memory should be released](https://rules.sonarsource.com/cpp/RSPEC-3584/) |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Avoid Performing Operations Inside Assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions should be used to detect logical errors inside code. The expression inside the assertion should not perform operations, it should only be used to make comparisons. |

| **Noncompliant Code** |
| --- |
| The expression inside this assertion assigns x with the value 12, rather than checking that x is equal to 12. |
| int x = 7;  int y = 5;  x = x + y  assert(x = 12); |

| **Compliant Code** |
| --- |
| The expression inside the assertion has been corrected to compare x with the value 12. |
| int x = 7;  int y = 5;  x = x + y;  assert(x == 12); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Proper use of assertions are essential for testing code, such as during unit tests. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 12.14 | - assignment in an assert statement |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Catch All Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When an exception is not caught, the program terminates prematurely. To prevent this, code should be added to catch all exceptions. This will handle any exceptions that the developer did not plan for. |

| **Noncompliant Code** |
| --- |
| While this code example does contain exception handling, it catches an exception other than the one thrown by the function. As such, the exception thrown by the function is not caught and the program terminates prematurely. |
| void function(){  throw std::runtime\_error(); }  int main(){  try{  function();  }  catch(std::logic\_error& e){  } } |

| **Compliant Code** |
| --- |
| This code contains an added exception handler that will catch any type of exception thrown. That way, even if the code throws an exception that the program does not account for, there is logic to handle it and the program will not terminate prematurely. |
| void function(){  throw std::runtime\_error(); }  int main(){  try{  function();  }  catch(std::logic\_error& e){  }  catch(...){  } } |

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

| **Principles(s):** Heed Compiler Warnings – Some compilers will generate warnings that exceptions may be unhandled.  Architect and Design for Security Policies – Preemptively planning to catch any exception is a way to design for security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 12.14 | - Unhandled exception specification when calling function foo() |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Avoid Comparing Signed and Unsigned Integers** |
| --- | --- | --- |
| **Data Types** | [STD-008-CPP] | When comparing signed and unsigned integers, the code may not work as expected. The signed integer could be implicitly cast to unsigned, leading to bugs. |

| **Noncompliant Code** |
| --- |
| This code could cause the signed int to be cast as unsigned int during the comparison. While the expected functionality is that the if statement is executed, when -1 is type cast to unsigned int it equals the max value of unsigned int. Therefore, the if statement is not executed. |
| unsigned int u 5;  int i -1;  if(i < u){  } |

| **Compliant Code** |
| --- |
| In the following example, the unsigned int is explicitly cast as a long. Long is used rather than int because the maximum value of unsigned int is greater than the maximum value of int. Since int and long are both signed types, the comparison works as expected. |
| unsigned int u 5;  int i -1;  if(i < static\_cast<long>(u)){  } |

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

| **Principles(s):** Heed Compiler Warnings – Some compilers will warn that comparing different types may lead to errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 12.14 | - dangerous sign conversion, when signed value can be negative |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close Files When They Are No Longer Needed** |
| --- | --- | --- |
| **Files** | [STD-009-CPP] | Having files open uses system resources. Files should be closed after they have been used. |

| **Noncompliant Code** |
| --- |
| This code opens a file to read text from, but never closes the file. |
| int main(){  string fileText;  ifstream f(filename.txt);  while(getline(f, fileText){  cout << fileText;  }  return 0; } |

| **Compliant Code** |
| --- |
| This code adds a statement to close the file when it is no longer in use. |
| int main(){  string fileText;  ifstream f(filename.txt);  while(getline(f, fileText){  cout << fileText;  }  f.close();  return 0; } |

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

| **Principles(s):** Architect and Design For Security Policies – As part of the design phase, whenever a file is opened a plan should also be made to close the file. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarLint | 8.2.1 | [Resource Should Be Closed](https://rules.sonarsource.com/cpp/RSPEC-2095/) |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Ensure You Do Not Divide By Zero** |
| --- | --- | --- |
| **Data Value** | [STD-010-CPP] | Dividing by zero is a runtime error that causes either undefined behavior or the code to exit prematurely. |

| **Noncompliant Code** |
| --- |
| The following code takes user input and uses it to perform division. There is the possibility that the user will input 0 for the denominator, resulting in a runtime error. |
| int a, b;  cin >> a >> b;  double result = a / b; |

| **Compliant Code** |
| --- |
| The compliant code checks for the denominator being zero before performing division. If b is zero, a runtime error exception is thrown that can be handled without terminating the program. |
| int a, b;  cin >> a >> b;  if(b == 0){  throw runtime\_error(“Dividing by zero attempted”); }  else{  double result = a / b;  } |

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

| **Principles(s):** Validate Input Data – This standard involves validating user input data to ensure that it does not cause a runtime error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 12.14 | - division with zero |  |
| SonarLint | 8.2.1 | [Zero should not be a possible denominator](https://rules.sonarsource.com/cpp/RSPEC-3949/) |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

There are several areas of the DevOps process where automation can be employed to enforce the security policy. During the build phase, compilers should use the highest level of warnings. This will automatically detect several potential issues with the code. During the verify and test phase, both static testing and unit testing can be employed. Static testing can point to problems that the compiler warnings overlooked. Several test beds of unit tests should be developed during this phase that will automatically determine that code is functioning as it should. These unit tests should be periodically run during the monitor and detect, respond, and maintain and stabilize phases to ensure that changes to the code have not altered the functionality.

### 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 | Low | Medium | Low | Low | 1 |
| STD-002-CPP | Low | High | Low | Medium | 2 |
| STD-003-CPP | Medium | Low | Medium | Medium | 3 |
| STD-004-CPP | High | High | High | High | 5 |
| STD-005-CPP | Medium | Low | Medium | Medium | 2 |
| STD-006-CPP | Low | High | Low | Medium | 1 |
| STD-007-CPP | Medium | High | Medium | High | 3 |
| STD-008-CPP | Low | Medium | Low | Low | 2 |
| STD-009-CPP | Medium | Low | Medium | Low | 2 |
| STD-010-CPP | Medium | High | Medium | Low | 3 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 refers to protecting data that exists in storage, such as on hard drives or in databases. Encrypting stored data protects the information it contains even if the data is leaked. Encryption at rest typically uses a symmetric encryption algorithm, such as the Advanced Encryption Standard. This policy should be employed for any sensitive data that is kept in storage. |
| Encryption in flight | Encryption in flight refers to protecting data as it is being transmitted between systems. This type of encryption is important, because data is most vulnerable when it is being transmitted. Unlike encryption at rest, encryption in flight uses asymmetric encryption. If the data is being transmitted over the internet, Transport Security Layer(TSL) should be employed. If the data is being transmitted between client and server, Secure Shell helps to protect the data. This policy should be applied whenever an outside source accesses an application, data is sent from the application to an outside source, and when sending emails. |
| Encryption in use | Encryption in use refers to protecting data as it is being used by an application. Data that is being processed in RAM typically needs to be decrypted, leaving it vulnerable. Adding encryption in use to encryption at rest and in flight allows for data to be encrypted for its entire lifecycle. Homomorphic encryption allows for operations such as addition and multiplication to be performed on encrypted data without decrypting it first. Another hardware solution to encryption in use is Trusted Execution Environments(TEE). These are areas in memory that are inaccessible from the rest of the system. The data is sent to the TEE, decrypted inside and processed, then encrypted again before leaving the TEE. Encryption in use should be utilized when working with highly sensitive data, such as financial or healthcare data, or when working on a cloud computing platform. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of users. This is typically done through the use of usernames and passwords, although systems can also use biometrics or tokens that generate a one-time passcode for every log in. Certificate based authentication is used to verify communications between devices. Multifactor authentication is common for systems that require higher levels of security, typically combining both password and token-based authentication. Authentication should be used whenever a user attempts to access the system. |
| Authorization | Authorization is the next step, after authentication. Authorization refers to giving users permission to access different functions of the system. In order for authorization to work, users must first be authenticated. A common authorization approach is Role Based Access Control, in which users are assigned roles and each role has different levels of authorization. The principle of least privilege states that users should only have authorization for the minimum number of functions they need. |
| Accounting | Accounting is the third part of the Triple-A Framework. Accounting is simply the process of logging user activity. When utilizing accounting, you store a record of when users log in, what changes they make, and what files they access. Accounting is important for security monitoring purposes. With accounting, you can recognize when unusual behavior occurs. For example, if a user logs in from a different IP address than they normally do, their account may have been compromised. Accounting is also legally required for certain businesses to comply with record-keeping requirements. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.2 | 8/11/2024 | Policy Creation | Adam Koenig |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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