**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 | At its core, do not trust any type of input. All types of input must be validated for range, length, format, and type. All input including input validation failures should be logged. |
| 1. Heed Compiler Warnings | It is recommended to use the highest level of warnings that the programming language and the type of compiler used should be used when compiling code. |
| 1. Architect and Design for Security Policies | A well architected and well designed security policy document must contain businesses requirements, its law, and regulations. It must address the potential risks involved when developing a system. |
| 1. Keep It Simple | “Complexity is the worst enemy of security”. Thus, when developing a system, it is important to keep things such as code, simple and easy to understand. The KISS principle emphasizes simplicity in design and system development which leads to more efficient development processes and better user experiences. |
| 1. Default Deny | All access to a system or network is denied by default except for individuals that must show they are authorized to access it. This protects the system against security threats |
| 1. Adhere to the Principle of Least Privilege | Individuals should only have access to what they need in order to perform 100% of their responsibilities, and no more. This enhances the protection of data, accounts being compromised, and from someone becoming and insider threat. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization is the practice of permanently deleting or destroying data purposely from storage devices in the hopes that it cannot be recovered. Some methods include physical destruction, data erasure, cryptographic erasure and data masking. |
| 1. Practice Defense in Depth | A practice in which multiple layers of security are placed all over a system. This ensures that if one security layer fails or is breached, others will still protect an organization’s data. Layers get progressively more secure and thus harder for an attacker to crack. This practice aims to slow down attacks rather than prevent them. |
| 1. Use Effective Quality Assurance Techniques | This principle ensures that the systems being created meets the standards and expectations of the end users. Methods are used to prevent defects and to improve the quality of the systems. |
| 1. Adopt a Secure Coding Standard | These are rules and guidelines that an organization must follow in order to detect, reduce, and eliminate errors that could cause security vulnerabilities when developing a system. These standards usually implement some of the above principles such as data validation and heap compiler warnings. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-C++] | Integers should not be used when performing arithmetic calculations that may yield a floating-point variable. This can lead to loss of information. |

| **Noncompliant Code** |
| --- |
| The code below shows multiplication and division operations on integers which are then converted to floating-point. Also, conversion to floating-point for the variables d and e happened after the arithmetic operations. |
| void myFunction(void) {  int a = 533;  int b = 6789;    float d = a / 7; /\*d is 76.0\*/  double e = b / 30; /\*e is 226.0\*/  } |

| **Compliant Code** |
| --- |
| The decimal error is eliminated by first storing the integers a and b in floating-point variables and then performing the arithmetic operations. This ensures at least one of the operands is a floating-point number and the arithmetic operations that follow are all performed on floating point numbers only. |
| void myFunction(void) {  short a = 533;  int b = 6789;  float c = a;  double d = b;    c /= 7; /\*d is 76.14286 \*/  d /= 30; /\*e is 226.3 \*/  } |

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

| **Principles(s):** Use floating-point variables to store whole numbers on areas where decimal arithmetic will be performed. This can be enforced by performing input validation to make sure a whole number is stored in a floating-point variable. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | P6 | L2 |

**Automation**

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

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-C++] | This standard is important for maintaining honest and usable data. Naming conventions for example, ensure that there is consistency for naming variables throughout the system. This can help identify how each data value is different. |

| **Noncompliant Code** |
| --- |
| The function below contains a long name that is very hard to read. The names of the variables inside the function are all using different types of naming conventions. This can introduce vulnerability issues like lack of clarity, and inconsistent naming. Also, the name of the function is too long. |
| Void functionforhowmanyyearsindogyears() {  Int Dog\_in\_human\_years;  int humanInDogYears;  int how-many-years-total;  } |

| **Compliant Code** |
| --- |
| The function name is shorter enough to convey what the function does. The names of the variables are also smaller. Convention for naming functions should start with the first word being lower case follow. |
| Void inDogYears() {  int dogToHumanYears;  int humanToDogYears;  int totalYears;  } |

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

| **Principles(s): Use noun or noun phrase names for classes and objects. Use verb or verb phrase names for methods or functions.** |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

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

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-C++] | String correctness is the concept of using the keyword const to prevent objects from getting changed. |

| **Noncompliant Code** |
| --- |
| In mathematics, PI is a constant. Here it is stored in a floating-point variable. It is not protected by const and thus it can be modified accidentally. |
| float = PI = 3.14159f;  float degrees;  float radians;  /\*Calculate radians\*/  radians = degrees \* PI / 180; |

| **Compliant Code** |
| --- |
| The keyword const is added to the variable PI to ensure that it is not change by the user or any function. |
| const float = PI = 3.14159f;  float degrees;  float radians;  /\*Calculate radians\*/  radians = degrees \* PI / 180; |

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

| **Principles(s):** Input validation should be used to check if constants are trying to be changed. Proper naming conventions can help identify constant variables from non-constant ones. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

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

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-Java] | An SQL injection exploits the password field of an account. The attack will create a boolean expression by “injecting” the condition (OR 1=1) into the password field. The boolean expression well then become true for all cases. |

| **Noncompliant Code** |
| --- |
| The noncompliant contains no way for the system to check if the ID and password entered in valid or not. This code authenticates the user without knowing the username ID and password by simply setting the username and password fields with '1' OR '1'='1'. |
| String query = "SELECT userName, balance FROM accounts"  + "WHERE userID=" + request.getParameter("userID") +  "and password='" + request.getParameter("Password") + "'";    try  {  Statement statement = connection.createStatement();  ResultSet rs = statement.executeQuery(query);  while (rs.next())  {  page.addTableRow(rs.getString("userName"),  rs.getFloat("balance"));  }  }  catch (SQLException e)  {} |

| **Compliant Code** |
| --- |
| SQL injection can be prevented by using prepared statements to create a parameterized query. This allows the database to distinguish between code and data. Statement.setInt() will throw an SQLexception error if it does not detect an integer from the user. |
| String query = "SELECT userName, balance "+  "FROM accounts WHERE userID = ?  and password = ?";    try {  PreparedStatement statement = connection.prepareStatement(query);  statement.setInt(1, request.getParameter("userID"));  ResultSet rs = statement.executeQuery();  while (rs.next())  {  page.addTableRow(rs.getString("userName"),  rs.getFloat("balance"));  }  } catch (SQLException e)  { ... } |

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

| **Principles(s):** Input validation should be taught as well as enforced it. The use of unit testing can help with making sure input validation is properly implemented. Configure proper error reporting and handling on web servers and in the code so that error messages are not sent to the client’s web browser. |
| --- |

**Threat Level**

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

**Automation**

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

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C] | There must be a way to control memory access rights on a physical computer. Memory protection prevents software from taking control of huge amounts of memory in the computer that can cause damage or affect other software that is currently being run on the computer. It also helps prevent the loss of saved data. |

| **Noncompliant Code** |
| --- |
| Memory leaks cause unintentional use of memory. Here the code fails to free a memory block containing data that is no longer needed. Nine of the 10 allocations of memory are lost because there is no way to free up space of the allocated memory blocks. |
| #include <stdlib.h>  #include <stdio.h>  #define LOOPS 10  #define MAXSIZE 256  int main(int argc, char \*\*argv)  {  int count = 0;  char \*pointer = NULL;  for(count=0; count<LOOPS; count++) {  pointer = (char \*)malloc(sizeof(char) \* MAXSIZE);  }  return count;  } |

| **Compliant Code** |
| --- |
| A simple fix is to add the free() function inside the for loop. It deallocates the memory that was previously allocated by the malloc() function. |
| #include <stdlib.h>  #include <stdio.h>  #define LOOPS 10  #define MAXSIZE 256  int main(int argc, char \*\*argv)  {  int count = 0;  char \*pointer = NULL;  for(count=0; count<LOOPS; count++) {  pointer = (char \*)malloc(sizeof(char) \* MAXSIZE);  }  free(pointer);  return count;  } |

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

| **Principles(s):** Free up memory form the heap when it is not needed. Use any garbage collection tools available. Terminate programs that are using too much memory. |
| --- |

**Threat Level**

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

**Automation**

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

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C++] | Assertions ensures code is robust and ensures it can handle unexpected situations |

| **Noncompliant Code** |
| --- |
| The conditional statement is not safe to use when doing parameter checking because if it fails, the program may continue to execute. |
| void myFunction(int x) {  if (x > 0)  {  //do something  }    else  {  std::cout << “x must be greater than 0”;  }  }  int main() {  myFunction(7);  return 0;  } |

| **Compliant Code** |
| --- |
| Bu using an assertion to check if x is greater than zero it ensures that the program terminates if this condition is not met. It also prints an error message. |
| #include <cassert>  void myFunction(int x) {  assert(x > 0 && "x must be greater than 0");  }  int main() {  myFunction(7);  return 0;  } |

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

| **Principles(s):** Use assertions extensively throughout the life cycle and even more during the testing phase. Assertions and exception handling are not the same. The team should be able to clearly differentiate between the two and use them appropriately. |
| --- |

**Threat Level**

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

**Automation**

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

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-C++] | Exceptions handle runtime errors or problems that a program encounters during execution time. This ensures code is easily maintainable and readable when debugging |

| **Noncompliant Code** |
| --- |
| One vulnerability of avoiding the use of exceptions is data leakage. Here there is no exceptions that when an error occurs, the delete might never be called and thus not actually deleting the data contain in importantInfo. |
| void myFunction()  {  myClass \* importantInfo = new myClass();  //Do something here  delete importantInfo;  } |

| **Compliant Code** |
| --- |
| Here an exception handling construct called try-catch is added to the function. The code in try is executed, if there is an error, the code will automatically execute the code in catch and then terminate the program. |
| void myFunction()  {  myClass \* importantInfo = 0;  try {  mine = new myClass();  // Do something here  }  catch(...) {  delete importantInfo;  throw;  }  delete importantInfo;  } |

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

| **Principles(s):** The team should be able to know the difference between exceptions and assertions. Exceptions should be used to check something that might happen. |
| --- |

**Threat Level**

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

**Automation**

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

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Code Complexity | [STD-008-C++] | Code should not be complex. Complex code can be hard to understand and debug. Comments should be added to explain methods, functions, classes. |

| **Noncompliant Code** |
| --- |
| There is a lot going on in the main method of the program. Comments are present but they do not explain the more important parts of the code. It is also too simple that it’s hard to understand what the variables mean. |
| #include <iostream>  int main() {  int x = 17;  int y = 2012;  std::string z = “October”;  //prints the date  std::cout << “Todays date is: “ << z << x << “, “ << y << std::endl;  } |

| **Compliant Code** |
| --- |
| Code is moved to a function out of main. More descriptive comments are added as well as better names for the variables. |
| #include <iostream>  //Outputs the current date  void currentDate()  {  int day = 17;  std::string month = “October”;  int year = 2012;  std::cout << “Todays Date is: “ << month << day << “, “ << year << std::endl;  }  Int main()  {  //Calls the currentDate() function to print the date00  currentDate();  //Terminates the program  Return 0;  } |

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

| **Principles(s):** Code should not be complex nor too simple. Follow naming conventions and stick to one. Communicate with the team to provide consistency when writing the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

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

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Naming | [STD-009-C++] | When naming pointers, they should have a lowercase p then the name to better identify a pointer. Also the \* should be placed to the name of the name and not the pointer type |

| **Noncompliant Code** |
| --- |
| It can hard to distinguished pointers from non-nonpointer variables, especially when needing to declare multiple variables in the same line. |
| int main()  {  int number = 24;  int\* ptr;  ptr = &number;    } |

| **Compliant Code** |
| --- |
| The \* is added to the name instead of the type. Also, the name of the pointer is more descriptive as opposed to just ptr. The n is added to emphasize that it is a pointer. |
| int main()  {  int number = 24;  int \*pNumber;  pNumber = &number;    } |

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

| **Principles(s):** Consistent naming convention for pointers should be included when deciding code complexity among the team. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

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

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Close Files | [STD-010-C++] | When working with files, the open() method should be matched with a close() method. Files should always be closed when they are done being used |

| **Noncompliant Code** |
| --- |
| The file remains open after the if statement. |
| #include <exception>  #include <fstream>  #include <string>    void myFunction(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  //do something  return;  }  std::terminate();  } |

| **Compliant Code** |
| --- |
| By adding an additional if statement, the code ensures the file is properly closed after being used or after it encounters an error. |
| #include <exception>  #include <fstream>  #include <string>    void myFunction(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  //do something  return;  }  file.close();  if (file.fail()) {  //do soemthing  }  std::terminate();  } |

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

| **Principles(s):** Provide file closing checker when opening files to ensure they are properly closed after they are done being used. Educate the team of improper file closing and how it leads to data loss. |
| --- |

**Threat Level**

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

**Automation**

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

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Likely | Medium | P6 | L2 |
| STD-002-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-003-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-004-CPP | Low | Probable | Medium | P4 | L3 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | High | Likely | High | P9 | L2 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-010-CPP | Medium | Unlikely | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Data not being used or stored should be encrypted at the database level. Also, backups for the data should be implemented in case of data loss. |
| Encryption at flight | Data that is being transferred from one location to another. Encryption keys can used to secure the data in motion as well as using protected tunnels like HTTPS or SSL/TLS |
| Encryption in use | It is data that is currently being worked on or being altered. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Ensure that the user is who they really are by asking the user for their login credentials. For an extra layer of protection, the user might need to provide further information (like two-step verification) to access more restricted data |
| Authorization | After the user has logged in, authorization should check what type of user he/she is and what data or resources they are allowed to work with. |
| Accounting | If a user is under a network or using the web-based application. Their activity should be monitored and logged. |

**\***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 | 10/11/2023 | Update Policies | Jonathan Ramirez |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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