**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data can mitigate and potentially avoid security risks associated with injection attacks (e.g. SQL injection, etc.) by preventing malicious input that may expose vulnerabilities. |
| 1. Heed Compiler Warnings | Ensuring compiler warnings are carefully scrutinized since it can provide valuable insights on potential deprecated functionalities related to hidden security issues. |
| 1. Architect and Design for Security Policies | Incorporating security policies and runbooks addressing access control and user authentications into early stage of code development leads to a more robust, secure, and resilient architectural system. |
| 1. Keep It Simple | Maintaining a security frame that is simple while robust so that if injections or attacks are detected, system logs are meaningful and easy to understand for guidance into issue mitigation. |
| 1. Default Deny | Minimizing broad public access to the system but instead involving configuration systems that accept permitted access with explicit actions. |
| 1. Adhere to the Principle of Least Privilege | Limiting user access to the “just-enough” level and grant or revoke access accordingly during routine audit to minimize potential damage caused by inappropriate access or access lingering for non-existent employees. |
| 1. Sanitize Data Sent to Other Systems | Reviewing data transferred to other systems to ensure sensitive data are properly redacted and the recipient intakes lightweighted, vulnerability-free data. |
| 1. Practice Defense in Depth | Implementing multi-layered security controls (e.g. firewall, encryption, etc.) to protect against a variety of attacks and keep system resiliency |
| 1. Use Effective Quality Assurance Techniques | Employing effective strategy (e.g. blue-green, canary, etc.) to ensure defensive measures are equipped in both active and passive environments. Executing periodic Disaster Recovery tests addresses hidden and unforeseen issues. |
| 1. Adopt a Secure Coding Standard | Following the best coding standards ensures adherence to security policies and overall consistency in security practices to reduce multifactorial scattered vulnerabilities. |

### 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** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not attempt to cv-qualify a reference type because it results in undefined behavior |

| **Noncompliant Code** |
| --- |
| Declares p to be a reference to a const-qualified char. The subsequent modification of p makes the program ill-formed |
| #include <iostream>    **void** f(**char** c) {  **const** **char** &p = c;    p = 'p'; // Error: read-only variable is not assignable    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| Removes the const qualifier |
| #include <iostream>    **void** f(**char** c) {  **char** &p = c;    p = 'p';    std::cout << c << std::endl;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard  Ensure to follow the best coding standards and adhere to security policies and overall consistency in security practices to reduce multifactorial scattered vulnerabilities. In this case, the const definition introduces a vulnerability that needs to be addressed to adopt secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube C/C++ Plugin | 4.10 | S3708 | SonarQube, like CppCheck, checks for vulnerabilities in the code and provides warning/error messages for the developers to revise |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | When dynamically checking for out-of-range values, checking must be performed before the cast expression to avoid operating on unspecified values. |

| **Noncompliant Code** |
| --- |
| Attempts to check whether a given value is within the range of acceptable enumeration values, but it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| Checks the value can be represented by the enumeration type before performing the conversion |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** Validate Input Data  Ensure to validate input data to mitigate and potentially avoid security risks associated with injection attacks by preventing malicious input that may expose vulnerabilities. In this case, the input data validation is placed after casting to the enum type, which causes unexpected behaviors |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP\_INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The std::basic\_string type uses the traits design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Passing a null pointer to these implementations/functions would result in dereferencing a null pointer |

| **Noncompliant Code** |
| --- |
| Std::string is created from the results of a call to std::getenv(), but because std::getenv() returns a null pointer on failure, this code below leads to undefined behavior |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The results from the call to std::getenv() are checked for null before the std::string object is constructed |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** Validate Input Data  This is another case of potentially having invalid input data that causes bugs. However, compared to the previous case, this case is more intense and critical because it results in random behaviors that in a production environment it may impact other properly-running processes. Therefore this is considered a high priority and criticality. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023b | CERT C++: STR51-CPP | Checks for string operations on null pointer |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-J] | SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query |

| **Noncompliant Code** |
| --- |
| This code permits a SQL injection attack by incorporating the unsanitized input argument username into the SQL command |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;    **class** Login {  **public** Connection getConnection() **throws** SQLException {      DriverManager.registerDriver(**new**              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  **return** DriverManager.getConnection(dbConnection);    }      String hashPassword(**char**[] password) {      // Create hash of password    }    **public** **void** doPrivilegedAction(String username, **char**[] password)  **throws** SQLException {      Connection connection = getConnection();  **if** (connection == **null**) {        // Handle error      }  **try** {        String pwd = hashPassword(password);          String sqlString = "SELECT \* FROM db\_user WHERE username = '"                           + username +                           "' AND password = '" + pwd + "'";        Statement stmt = connection.createStatement();        ResultSet rs = stmt.executeQuery(sqlString);    **if** (!rs.next()) {  **throw** **new** SecurityException(            "User name or password incorrect"          );        }          // Authenticated; proceed      } **finally** {  **try** {          connection.close();        } **catch** (SQLException x) {          // Forward to handler        }      }    }  } |

| **Compliant Code** |
| --- |
| This code validates the length of the username argument, preventing an attacker from submitting a long username |
| **public** **void** doPrivilegedAction(    String username, **char**[] password  ) **throws** SQLException {    Connection connection = getConnection();  **if** (connection == **null**) {      // Handle error    }  **try** {      String pwd = hashPassword(password);        // Validate username length  **if** (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString);      stmt.setString(1, username);      stmt.setString(2, pwd);      ResultSet rs = stmt.executeQuery();  **if** (!rs.next()) {  **throw** **new** SecurityException("User name or password incorrect");      }        // Authenticated; proceed    } **finally** {  **try** {        connection.close();      } **catch** (SQLException x) {        // Forward to handler      }    }  } |

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

| **Principles(s):** Validate Input Data  SQL injection can not only impact overall performance but also contaminate database. Constant rehydration and backlog comparisons need to be carried out to ensure database’s integrity |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | JAVA.IO.INJ.SQL | SQL Injection |
| Fortify | 1.0 | HTTP\_Response\_splitting  SQL\_Injection\_Persistence  SQL\_Injection | Checks for SQL Injection |
| SonarQube | 9.9 | S2077  S3649 | Executing SQL queries is security-sensitive |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not store an already-owned pointer value in an unrelated smart pointer** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not create an unrelated smart pointer object with a pointer value that is owned by another smart point object |

| **Noncompliant Code** |
| --- |
| Two unrelated smart pointed are constructed from the same underlying point value |
| #include <memory>    **void** f() {  **int** \*i = **new** **int**;    std::shared\_ptr<**int**> p1(i);    std::shared\_ptr<**int**> p2(i);  } |

| **Compliant Code** |
| --- |
| Std::shared\_ptr objects are related to one another through copy construction. It also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local variable |
| #include <memory>    **void** f() {    std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();    std::shared\_ptr<**int**> p2(p1);  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems  Reviewing data transferred to other systems to ensure sensitive data are properly redacted and the recipient intakes lightweighted, vulnerability-free data. In this case, during struct, the data is complicated by constructing two separate pointers pointing to the same value. When one pointer gets deleted, the other will convert to a null pointer as well |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: MEM56-CPP | Checks for use of already-owned pointers |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Static assertion resolves the problem with the runtime assert() which has limitations and is useful only for identifying incorrect assumptions and not for runtime error checking |

| **Noncompliant Code** |
| --- |
| Uses assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| A preprocessor conditional statement is used for assertions involving only constant expressions |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| Employing effective strategy (e.g. blue-green, canary, etc.) to ensure defensive measures are equipped in both active and passive environments. Executing periodic Disaster Recovery tests addresses hidden and unforeseen issues. In this case, assert() could be used in a pre-release environment, but should be replaced by diagnostic messages if being released to a production environment to avoid hardcoded values |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ÉCLAIR | 1.2 | CC2.DCL03 | Detect violations by looking for assert() calls |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions thrown by an application must be caught by a matching exception handler |

| **Noncompliant Code** |
| --- |
| Neither f() nor main() catch exceptions thrown by throwing\_func() |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| The main entry point handles all exceptions which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard  In all program codes, exceptions should be handled fully and carefully. When an exception is thrown, it must be caught either in the current block, or in a subsequent code block that has the function running. Improper exception handling could result in compilation and runtime errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: ERR60-CPP | Checks for throwing exception object in copy constructor |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Gracefully handle self-copy assignment** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | User-provided copy operators must properly handle self-copy assignment |

| **Noncompliant Code** |
| --- |
| The copy assignment operator does not protect against self-copy assignment |
| #include <new>    **struct** S { S(**const** S &) noexcept; /\* ... \*/ };    **class** T {  **int** n;    S \*s1;    **public**:    T(**const** T &rhs) : n(rhs.n), s1(rhs.s1 ? **new** S(\*rhs.s1) : nullptr) {}    ~T() { **delete** s1; }      // ...      T& operator=(**const** T &rhs) {      n = rhs.n;  **delete** s1;      s1 = **new** S(\*rhs.s1);  **return** \***this**;    }  }; |

| **Compliant Code** |
| --- |
| Self-copy assignment is guarded against by testing whether the given parameter is the same as “this” |
| #include <new>    **struct** S { S(**const** S &) noexcept; /\* ... \*/ };    **class** T {  **int** n;    S \*s1;    **public**:    T(**const** T &rhs) : n(rhs.n), s1(rhs.s1 ? **new** S(\*rhs.s1) : nullptr) {}    ~T() { **delete** s1; }      // ...      T& operator=(**const** T &rhs) {  **if** (**this** != &rhs) {        n = rhs.n;  **delete** s1;  **try** {          s1 = **new** S(\*rhs.s1);        } **catch** (std::bad\_alloc &) {          s1 = nullptr; // For basic exception guarantees  **throw**;        }      }  **return** \***this**;    }  }; |

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

| **Principles(s):** Keep It Simple  Maintaining a security frame that is simple while robust so that if injections or attacks are detected, system logs are meaningful and easy to understand for guidance into issue mitigation. In this case, the noncompliant code was trying to condense itself with lower line count but ignore possible nullptr errors. Therefore, when programing, keep the code simple and easy-to-understand. When reviewing the code, developers can easily see where the issue occurs and can implement simple fix, rather than spending time trying to understand what the condensed code does. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: OOP54-CPP | Checks for copy assignment operators where self-assignment is not tested |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input and Output | [STD-009-CPP] | A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended |

| **Noncompliant Code** |
| --- |
| The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate handler called by std::terminate() is std::abort() which does not call destructors |
| #include <exception>  #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {    std::fstream file(fileName);  **if** (!file.is\_open()) {      // Handle error  **return**;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| Std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed |
| #include <exception>  #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {    std::fstream file(fileName);  **if** (!file.is\_open()) {      // Handle error  **return**;    }    // ...    file.close();  **if** (file.fail()) {      // Handle error    }    std::terminate();  } |

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

| **Principles(s):** Practice Defense in Depth  When a file is not closed, it will stay open in the background not only consuming resources but also becoming vulnerable to any attacks. Therefore, developers need to keep in mind to practice defense in depth and be vigilant at all time to be prepared for any unforeseen issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++: FIO51-CPP | Checks for resource leak |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Ensure actively held locks are released on exceptional conditions** |
| --- | --- | --- |
| Concurrency | [STD-010-CPP] | The throwing of an exception must not allow a mutex to remain locked indefinitely |

| **Noncompliant Code** |
| --- |
| The code unlocks the mutex when it is finished, but if an exception occurs, the mutex will remain locked indefinitely |
| #include <mutex>  void manipulate\_shared\_data(std::mutex &pm) {  pm.lock();  // Perform work on shared data.  pm.unlock();  } |

| **Compliant Code** |
| --- |
| Catches any exceptions thrown when performing work on the shared data and unlocks the mutex before rethrowing the exception |
| #include <mutex>  void manipulate\_shared\_data(std::mutex &pm) {  pm.lock();  try {  // Perform work on shared data.  } catch (...) {  pm.unlock();  throw;  }  pm.unlock(); // in case no exceptions occur  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques  When using mutex, make sure it’s locked/unlocked when starting/finishing the code block. Catch/Throw exceptions when necessary to prevent it from being permanently locked |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | CONCURRENCY.LOCK.NOUNLOCK | Missing Lock Release |

### 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.

Green Pace’s establishment of the two-tier staging is very helpful and effective to address security concerns. In the Pre-production environment, unit testing, warnings/errors check should be implemented as part of the development stage. Automations such as add-ons to IDEs could be implemented to reduce human errors. When it comes to a QA environment, multi-staged testing could be implemented from single-point user testing to multi-user batch testing. Specific sub-environments such as UAT could be introduced to ensure that not only does the application function normally but also does it stay secure and vulnerable-free when introduced to a live environment. Post release, monitors and metrics should be set up for production environments to detect any outliers or anomalies happening when different users (with good intent or not) interact with the application.

### 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 | Unlikely | Low | P3 | L3 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-J | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-C | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Probable | High | P2 | L3 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Low | Probable | Low | P6 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest represents data encryption that is stored on-premise. For example, data that is stored in a database needs encryption in rest. When data is pushed to a database, instead of storing them in pure string format, all data should be following a specific encryption protocol to ensure such data is not visible or decryptable by the general public. This is to ensure that all on-prem data is stored in a safe and secure place that if hackers find a way in, the encrypted data is still not decipherable by the hackers. |
| Encryption at flight | Encryption at flight means data encryption during transit. For example, when sending an email with sensitive information like username and password, such email needs to be encrypted. Traditionally, such encryption generates a personal key on the sender’s device and a decryption key on the receiver’s device. The email in transit is encrypted in a way that only the devices with the paired keys can decrypt the message. This policy applies to ensure that in case of sensitive information needed to be sent from one entity to another, it does not get hacked during transit. |
| Encryption in use | Encryption in use ensures that once the authorized entity receives and decrypted an encrypted piece of data, such data will remain encrypted like processing plaintext. This is to ensure that if hacker somehow hacks into this entity’s device, without the proper key or authorization, the data is still uninterpretable by the hacker |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication provides the means to identify a user with proper credentials. Once the credential is authenticated, the user can be granted certain access. Authentication serves as the entrance ward to ensure anyone who is coming in has the proper and correct credentials. |
| Authorization | Authorization checks whether a user has access to perform certain actions once the user passes authentication. This is to ensure that only authorized personnel have access to certain areas of a given application. For example, an admin will have a pan-access to all functionalities of an application, while a ReadOnly user may only view the description page. |
| Accounting | Accounting links with metrics and monitoring because it measures the resources users consume during access. For example, once a user logs in, the session should be persisted until the user exits or logs out. Meanwhile, if the user performs regular tasks, then accounting metrics should look normal; however, if DDoS is detected, then anomalies will be detected through accounting. Accounting exists to ensure that no user is abusing or misusing the granted access to perform illegitimate tasks. |

**\***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 | 02/18/2024 | Project 1 | Allan Lee | [Insert text.] |
| [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 |