**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 | Ensure that all input data is validated to prevent malicious code or unexpected data from causing vulnerabilities. |
| 1. Heed Compiler Warnings | Pay attention to compiler warnings as they can often indicate potential security vulnerabilities or coding errors. |
| 1. Architect and Design for Security Policies | Design software with security policies in mind, ensuring that access controls and other security measures are integrated into the architecture. |
| 1. Keep It Simple | Simplicity in design and implementation can reduce the potential for security vulnerabilities. |
| 1. Default Deny | Configure systems to deny access by default, only allowing access to resources that are explicitly permitted. |
| 1. Adhere to the Principle of Least Privilege | Grant users the minimum level of access necessary for them to perform their duties. |
| 1. Sanitize Data Sent to Other Systems | Ensure that data sent to other systems is sanitized and validated to prevent injection attacks. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls to protect against various types of attacks. |
| 1. Use Effective Quality Assurance Techniques | Employ testing and quality assurance processes to identify and fix security vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Follow a secure coding standard, such as those provided by CERT or OWASP, to ensure that code is written with security in mind. |

### 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** | EXP00-CPP | Prefer using int for integer values unless a smaller type is explicitly needed. |

| **Noncompliant Code** |
| --- |
| Using smaller integer types unnecessarily can lead to overflow and performance issues. |
| short count = 32767;  count += 1; // Overflow |

| **Compliant Code** |
| --- |
| Using int for general-purpose integer values. |
| int count = 32767;  count += 1; // No overflow |

| **Principles(s):** Using int reduces the risk of overflow and improves performance. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | IntegerTypeChecker | Detects unnecessary use of smaller integer types. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | DCL52-CPP | Validate input data values to ensure they are within expected ranges. |

| **Noncompliant Code** |
| --- |
| Failing to validate input data can lead to incorrect behavior or security vulnerabilities. |
| #include <iostream>  void setAge(int age) {  // No validation  std::cout << "Age is: " << age << std::endl;  }  int main() {  setAge(-5); // Invalid age  return 0;  } |

| **Compliant Code** |
| --- |
| Validating input data to ensure it is within the expected range. |
| #include <iostream>  #include <stdexcept>  void setAge(int age) {  if (age < 0 || age > 150) {  throw std::out\_of\_range("Age must be between 0 and 150.");  }  std::cout << "Age is: " << age << std::endl;  }  int main() {  try {  setAge(25); // Valid age  setAge(-5); // Invalid age  } catch (const std::out\_of\_range& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

| **Principles(s):** Ensures input data is within expected ranges, preventing incorrect behavior and potential security issues. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | InputValidationChecker | Detects lack of input validation for data values. |

#### Coding Standard 3

| **String Correctness** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STR51-CPP | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| Modifying string literals can cause undefined behavior. |
| char\* str = "Hello";  str[0] = 'h'; |

| **Compliant Code** |
| --- |
| Using a character array or std::string for modifiable strings. |
| std::string str = "Hello";  str[0] = 'h'; |

| **Principles(s):** Prevents modification of read-only memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | StringLiteralModificationChecker | Detects attempts to modify string literals. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | SQL51-CPP | Prevent SQL Injection by using parameterized queries. |

| **Noncompliant Code** |
| --- |
| Using string concatenation to build SQL queries can lead to SQL injection vulnerabilities. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  void executeQuery(sqlite3\* db, const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'";  char\* errMsg = nullptr;  if (sqlite3\_exec(db, query.c\_str(), nullptr, nullptr, &errMsg) != SQLITE\_OK) {  std::cerr << "SQL error: " << errMsg << std::endl;  sqlite3\_free(errMsg);  }  }  int main() {  sqlite3\* db;  sqlite3\_open("example.db", &db);  executeQuery(db, "Alice' OR '1'='1");  sqlite3\_close(db);  return 0;  } |

| **Compliant Code** |
| --- |
| Using parameterized queries to prevent SQL injection. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  void executeQuery(sqlite3\* db, const std::string& userInput) {  sqlite3\_stmt\* stmt;  const char\* query = "SELECT \* FROM users WHERE name = ?";    if (sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr) != SQLITE\_OK) {  std::cerr << "Failed to prepare statement" << std::endl;  return;  }    if (sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC) != SQLITE\_OK) {  std::cerr << "Failed to bind parameter" << std::endl;  sqlite3\_finalize(stmt);  return;  }  while (sqlite3\_step(stmt) == SQLITE\_ROW) {  std::cout << "User found: " << sqlite3\_column\_text(stmt, 0) << std::endl;  }  sqlite3\_finalize(stmt);  }  int main() {  sqlite3\* db;  sqlite3\_open("example.db", &db);  executeQuery(db, "Alice");  sqlite3\_close(db);  return 0;  } |

| **Principles(s):** Prevents SQL injection attacks by ensuring user input is properly handled. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Analysis Tool | 1.0 | SQLInjectionChecker | Detects potential SQL injection vulnerabilities by analyzing query construction and parameter binding. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM51-CPP | Properly deallocate dynamically allocated resources. |

| **Noncompliant Code** |
| --- |
| Failing to deallocate resources leads to memory leaks. |
| void exampleFunction()  {  int\* arr = new int[10];  // Memory leak, arr not deleted  } |

| **Compliant Code** |
| --- |
| Ensuring resources are properly deallocated. |
| void exampleFunction() {  int\* arr = new int[10];  // Do something with arr  delete[] arr;  } |

| **Principles(s):** Ensures proper cleanup of dynamically allocated resources. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | MemoryLeakChecker | Detects potential memory leaks. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | ERR52-CPP | Use assertions to document and validate assumptions. |

| **Noncompliant Code** |
| --- |
| Failing to use assertions to validate assumptions can lead to undetected errors during development. |
| #include <iostream>  int divide(int a, int b) {  // No assertion to check if b is zero  return a / b;  }  int main() {  int result = divide(10, 0); // Undefined behavior, division by zero  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using assertions to validate assumptions, ensuring issues are detected during development. |
| #include <iostream>  #include <cassert>  int divide(int a, int b) {  assert(b != 0 && "Division by zero is undefined.");  return a / b;  }  int main() {  int result = divide(10, 2); // Valid operation  std::cout << "Result: " << result << std::endl;  result = divide(10, 0); // Assertion failure, detected during development  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Principles(s):** Ensures that assumptions are validated during development, reducing the likelihood of runtime errors in production. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | AssertionChecker | Detects missing assertions for validating assumptions and critical conditions. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR53-CPP | Use specific exceptions instead of generic exceptions. |

| **Noncompliant Code** |
| --- |
| Catching generic exceptions makes it harder to determine the specific error that occurred and how to handle it appropriately. |
| #include <iostream>  #include <stdexcept>  void exampleFunction() {  try {  throw std::runtime\_error("An error occurred");  } catch (...) {  std::cerr << "Caught an exception" << std::endl;  // No specific handling  }  } |

| **Compliant Code** |
| --- |
| Catching specific exceptions allows for more precise error handling and debugging. |
| #include <iostream>  #include <stdexcept>  void exampleFunction() {  try {  throw std::runtime\_error("An error occurred");  } catch (const std::runtime\_error& e) {  std::cerr << "Caught a runtime error: " << e.what() << std::endl;  // Handle runtime error specifically  } catch (const std::exception& e) {  std::cerr << "Caught an exception: " << e.what() << std::endl;  // Handle other types of standard exceptions  } catch (...) {  std::cerr << "Caught an unknown exception" << std::endl;  // Handle any other types of exceptions  }  } |

| **Principles(s):** Ensures that exceptions are handled precisely and appropriately, improving the robustness and clarity of error handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | ExceptionTypeChecker | Detects usage of generic exception handlers and recommends catching specific exception types. |

#### 

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Efficiency | EXP02-CPP | Use static\_cast rather than dynamic\_cast to downcast when the type of the object is known. |

| **Noncompliant Code** |
| --- |
| Use static\_cast rather than dynamic\_cast to downcast when the type of the object is known. |
| class Base {};  class Derived : public Base {};  void exampleFunction(Base\* base) {  Derived\* derived = dynamic\_cast<Derived\*>(base);  // Use derived  } |

| **Compliant Code** |
| --- |
| Using static\_cast for known types. |
| class Base {};  class Derived : public Base {};  void exampleFunction(Base\* base) {  Derived\* derived = static\_cast<Derived\*>(base);  // Use derived  } |

| **Principles(s):** static\_cast is more efficient and indicates certainty in type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | DowncastChecker | Detects unnecessary use of dynamic\_cast. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integrity | OOP51-CPP | Do not slice polymorphic objects. |

| **Noncompliant Code** |
| --- |
| Slicing a polymorphic object slices off the derived parts. |
| class Base {  public:  virtual void print() const {  std::cout << "Base" << std::endl;  }  };  class Derived : public Base {  public:  void print() const override {  std::cout << "Derived" << std::endl;  }  };  void exampleFunction(Base b) {  b.print();  } |

| **Compliant Code** |
| --- |
| Using pointers or references to avoid slicing. |
| class Base {  public:  virtual void print() const {  std::cout << "Base" << std::endl;  }  };  class Derived : public Base {  public:  void print() const override {  std::cout << "Derived" << std::endl;  }  };  void exampleFunction(const Base& b) {  b.print();} |

| **Principles(s):** Integrity: Preserves the full type information and behavior of polymorphic objects. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | ObjectSlicingChecker | Detects object slicing scenarios. |

#### 

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Safety | STR51-CPP | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| Modifying string literals can cause undefined behavior. |
| char\* str = "Hello";  str[0] = 'h'; |

| **Compliant Code** |
| --- |
| Using a character array or std::string for modifiable strings. |
| std::string str = "Hello";  str[0] = 'h'; |

| **Principles(s):** Prevents modification of read-only memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P1 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tool | 1.0 | StringLiteralModificationChecker | Detects attempts to modify string literals. |

### 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.

The provided DevSecOps diagram illustrates the integration of security practices into the DevOps workflow, emphasizing the importance of security at every stage of the software development lifecycle. Here's how automation can be used to enforce and ensure compliance with the standards defined in this policy, based on the diagram:

### ****Pre-production Phase****

#### ****Assess and Plan:****

* **Automation Task:** Utilize automated threat modeling and compliance tools to assess the threat landscape and regulatory changes. Tools like OWASP Threat Dragon can be integrated to automate threat modeling.
* **Implementation:** Regularly scheduled scans and assessments can be set up to run automatically, generating reports and alerts for any new threats or compliance issues.

#### ****Design:****

* **Automation Task:** Integrate security testing into the design phase with automated tools that follow best practices, such as OWASP standards.
* **Implementation:** Use tools like Fortify or Veracode for static application security testing (SAST) to automatically review code against security best practices during the design phase.

#### ****Build:****

* **Automation Task:** Ensure that the build process includes automated security checks and that secure, signed source code repositories are used.
* **Implementation:** Implement Continuous Integration (CI) tools like Jenkins or GitLab CI to automatically run security tests on each build, and ensure that only trusted, signed code is used in the build process.

#### ****Verify and Test:****

* **Automation Task:** Automate vulnerability scanning and security testing.
* **Implementation:** Integrate Dynamic Application Security Testing (DAST) tools like OWASP ZAP or Burp Suite into the CI/CD pipeline to automatically scan for vulnerabilities and perform security testing on the application.

### ****Production Phase****

#### ****Transition and Health Check:****

* **Automation Task:** Automate the deployment and security configuration checks.
* **Implementation:** Use Infrastructure as Code (IaC) tools like Terraform or Ansible to automatically configure and deploy environments with predefined security settings. Automated penetration testing tools can be scheduled to run at specific intervals.

#### ****Monitor and Detect:****

* **Automation Task:** Set up automated monitoring and logging.
* **Implementation:** Use Security Information and Event Management (SIEM) systems like Splunk or ELK Stack to automatically collect, analyze, and alert on security events. Integrate intrusion detection systems (IDS) to continuously monitor for threats.

#### ****Respond:****

* **Automation Task:** Automate incident response procedures.
* **Implementation:** Implement Security Orchestration, Automation, and Response (SOAR) tools to automate the response to security incidents. This includes blocking attacks, turning off compromised services, and rolling back to previous states.

#### ****Maintain and Stabilize:****

* **Automation Task:** Automate the assessment of security baselines and the stabilization process.
* **Implementation:** Use configuration management tools like Puppet or Chef to automatically ensure systems are in compliance with security baselines. Schedule regular assessments and automated remediation of any deviations from the baseline.

### ****Continuous Improvement:****

* Regularly update and improve the automation processes based on feedback and evolving security standards.
* Integrate new security tools and practices as they become available, ensuring that the DevSecOps pipeline remains effective and up-to-date.

### 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 |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | EXP00-CPP |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | |  | | --- | | Critical |  |  | | --- | |  | |
| |  | | --- | | DCL52-CPP |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | |  | | --- | | Low |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | Moderate |
| |  | | --- | | STR51-CPP |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | |  | | --- | | High |  |  | | --- | |  | | Low | |  | | --- | | High |  |  | | --- | |  | | Critical |
| |  | | --- | | SQL51-CPP |  |  | | --- | |  | | High | High | |  | | --- | | Medium |  |  | | --- | |  | | High | |  |  | | --- | --- | | |  | | --- | | Critical | | |
| |  | | --- | | MEM51-CPP |  |  | | --- | |  | | High | |  | | --- | | Medium |  |  | | --- | |  | | Medium | High | |  |  | | --- | --- | | |  | | --- | | Critical | | |
| |  | | --- | | ERR52-CPP |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | |  | | --- | | Medium |  |  | | --- | |  | | Moderate |
| |  | | --- | | ERR53-CPP |  |  | | --- | |  | | Medium | Medium | |  | | --- | | Low | | Medium | Moderate |
| |  | | --- | | EXP02-CPP |  |  | | --- | |  | | |  | | --- | | High | | Medium | Medium | |  | | --- | | High | | |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | | |  | | --- | | Critical | | | |
| |  | | --- | | OOP51-CPP |  |  | | --- | |  | | Medium | Medium | Low | |  | | --- | | Medium | | Moderate |
| |  | | --- | | STR51-CPP |  |  | | --- | |  | | |  | | --- | | High | | |  | | --- | | High | | Medium | |  | | --- | | High | | Critical |

### 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 | **What:** Protects stored data by encrypting it on physical or virtual storage devices.  **How:** Data is encrypted using algorithms like AES-256 before being saved to storage.  **Why:** Ensures data is protected from unauthorized access if storage media is compromised. |
| Encryption in flight | **What:** Protects data as it is transmitted over networks.  **How:** Data is encrypted using secure protocols such as TLS during transmission.  **Why:** Ensures data integrity and confidentiality during transmission over public or internal networks. |
| Encryption in use | **What:** Ensures data integrity and confidentiality during transmission over public or internal networks.  **How:** Techniques like homomorphic encryption and secure enclaves encrypt data during processing.  **Why:** Ensures data remains secure from unauthorized access during active use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **What:** The process of verifying the identity of users, devices, or applications.  **How:** Users log in using credentials (e.g., passwords, biometrics, MFA). Devices and applications use certificates or tokens.  **Why:** Ensures only authorized individuals and systems can access Green Pace's resources, protecting sensitive data from unauthorized access. |
| Authorization | **What:** Determines what authenticated users, devices, or applications are allowed to do.  **How:** Access is granted based on roles or policies, defining permissions for data and system functions (e.g., read, write, execute).  **Why:** Ensures users and systems can only perform actions they are permitted to, preventing unauthorized access to sensitive data and functions. |
| Accounting | **What:** Tracking and logging user activities for auditing and compliance purposes.  **How:** Logs are generated for user logins, database changes, addition of new users, user levels of access, and files accessed.  **Why:** Ensures accountability, helps detect unauthorized access and activities, and facilitates forensic analysis in case of security incidents. |

**\***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.

**Coding Standard 1: EXP00-CPP**

* **Principles:** 1 (Validate Input Data), 2 (Heed Compiler Warnings), 7 (Sanitize Data Sent to Other Systems)

**Coding Standard 2: DCL52-CPP**

* **Principles:** 3 (Architect and Design for Security Policies), 6 (Adhere to the Principle of Least Privilege)

**Coding Standard 3: STR51-CPP**

* **Principles:** 1 (Validate Input Data), 10 (Adopt a Secure Coding Standard)

**Coding Standard 4: SQL51-CPP**

* **Principles:** 1 (Validate Input Data), 7 (Sanitize Data Sent to Other Systems), 8 (Practice Defense in Depth)

**Coding Standard 5: MEM51-CPP**

* **Principles:** 1 (Validate Input Data), 9 (Use Effective Quality Assurance Techniques)

**Coding Standard 6: ERR52-CPP**

* **Principles:** 4 (Keep It Simple), 9 (Use Effective Quality Assurance Techniques)

**Coding Standard 7: ERR53-CPP**

* **Principles:** 4 (Keep It Simple), 9 (Use Effective Quality Assurance Techniques)

**Coding Standard 8: EXP02-CPP**

* **Principles:** 3 (Architect and Design for Security Policies), 9 (Use Effective Quality Assurance Techniques)

**Coding Standard 9: OOP51-CPP**

* **Principles:** 3 (Architect and Design for Security Policies), 6 (Adhere to the Principle of Least Privilege)

**Coding Standard 10: STR51-CPP**

* **Principles:** 1 (Validate Input Data), 7 (Sanitize Data Sent to Other Systems)

## 

## 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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [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 |