**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 | Ensures all the input data is checked for validity and accuracy. This can help reduce the number of attacks such as SQL Injection and Buffer Overflow. Which helps prevent malicious inputs from compromising the security. |
| 1. Heed Compiler Warnings | This addresses compiler warnings to ensure that any potential issues in the code are caught promptly and early. If these warnings are ignored it can lead to vulnerabilities in the code that can compromise the entire system. |
| 1. Architect and Design for Security Policies | Software architecture and design must be considered when implementing any security policies. If security policies are used during this phase, it will allow you to create a secure foundation which in turn will make your system a lot more secure. |
| 1. Keep It Simple | When a system is more complex it makes it more difficult to secure. More simplistic systems are easier to audit and protect against any vulnerabilities as well as making it a lot easier to understand. Using this principle can reduce errors in coding and use. |
| 1. Default Deny | Access to the any system should be denied by default until the user is granted specific access. This allows for the prevention of unauthorized access to the system and only gives access to users when the conditions of the protection scheme are verified. |
| 1. Adhere to the Principle of Least Privilege | Users and the system should be given the minimum level of access needed to perform their tasks. This can help reduce the risk of abuse and compromise of data due to excessive permissions. |
| 1. Sanitize Data Sent to Other Systems | When data is sanitized before being sent to other systems can help prevent SQL Injection attacks. This checks the data being sent for any potential problems or issues before being sent to another system which could open the system to unwanted attacks. |
| 1. Practice Defense in Depth | Multiple layers of defense are recommended to allow for backups if one or more of your defenses are penetrated. This can help reduce damage and exploits to the system. Examples would be a firewall, authentication, and encryption. |
| 1. Use Effective Quality Assurance Techniques | Performing rigorous testing of the system and QA techniques can help to identify any potential vulnerabilities in the system while also ensuring the system is robust and secure before it is deployed. It is also good practice to have security reviews internal as well as external to help identify any issues. |
| 1. Adopt a Secure Coding Standard | When using secure coding standards, you can make sure that there is consistency across all the code which can reduce vulnerabilities that can be caused by bad coding techniques. This can help to ensure that your system is secure from the foundation up. |

### 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-CPP] | Data types used correctly to prevent unintended behavior or any vulnerabilities due to the incorrect handling of types. |

| **Noncompliant Code** |
| --- |
| Wrong data types are used for the variables which could possibly lead to unexpected results or security issues/vulnerabilities. |
| int y = 10.5; // inferred conversion from floating-point value to integer.  char a = 300; // 300 exceeds the char limit which causes overflow. |

| **Compliant Code** |
| --- |
| Using the correct data types here we can avoid such issues as data loss and overflow which in turn enhances reliability and security. |
| Double y = 10.5; // Correct type for decimal number.  Int a = 300; // Correct type to be in the valid range of the value. |

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

| **Principles(s):**  **1. Validate Input data** - Utilizing the correct data types helps prevent incorrect inputs, which minimizes the chances of buffer overflows and related security vulnerabilities.  **2. Heed Compilier Warnings** - Compiler alerts often indicate possible problems with type conversions, allowing developers to resolve these issues early in the coding process.  **8. Practice Defense in Depth** - Strict data type enforcement serves as a protective measure within a broader defense-in-depth security approach. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | C | Finds any non-compliant type usage and overflow vulnerabilities. |
| Clang Static Analyzer | 12.0.1 | UndefinedBehaviorSanitizer | Finds any issues related to the type conversions and overflow. |
| Cppcheck | 2.8 | Type Conversion | Analyzes the code to find any unsafe type conversions or implicit data type changes |
| Coverity | 2024.1 | CERT.CERT-FLP34-C | Searches for floating-point values and type conversion vulnerabilities. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Implements protection against incorrect manipulation or assignment of data values to help prevent overflow, underflow, or potential security risks. |

| **Noncompliant Code** |
| --- |
| This code fails to validate the data boundaries which in turn can result in overflow or underflow. |
| int a = INT\_MAX; // Assigns the largest possible value to int.  a += 1; // Results in overflow. |

| **Compliant Code** |
| --- |
| Validations are implemented to ensure the data values remain in safe boundaries to prevent overflow. |
| int a = INT\_MAX;  if (a < INT\_MAX) {  a += 1; // Increment only when it is safe to do so.  }  else {  //Handle overflow case.  } |

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

| **Principles(s):**  **8. Practice Defense in Depth**-Adding checks for data integrity acts as a security layer against unintended data modifications.  **9. Use Effective Quality Assurance Techniques**-Consistent testing helps to detect cases of data overflow making sure protective measures are in place correctly.  **1. Validate Input Data**-This ensures data remains within allowable limits which helps to avoid overflow and underflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | C | Finds instances where the manipulation of data values can cause overflows. |
| Cppcheck | 2.8 | Integer Handling | Analyzes the code to find any possible places for underflow, overflow, or integer operations that are not safe. |
| Clang Static Analyzer | 12.0.1 | Integer Overflow | Finds any potential underflow and overflow situations with integers. |
| Coverity | 2024.1 | CERT.INT30-C | Checks for overflow and underflow in the data manipulations and assignments. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Helps prevent problems like buffer overflow or injection attacks that arise from improper string handling. |

| **Noncompliant Code** |
| --- |
| This code fails to correctly terminate the string which can lead to buffer overflow or unpredictable behavior. |
| char str[5];  strcpy(str, “Hello, world!”); // Buffer overflow occurs since the array size is insufficient. |

| **Compliant Code** |
| --- |
| This code safely manages the string preventing overflow by restricting the number of character being copied. |
| char str[5];  strcpy(str, “Hello, world!”, sizeof(str) – 1); // Limit the copy to the buffer size.  Str[4] = ‘\0’; // Manually add the null termination. |

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

| **Principles(s):**  **4. Keep It Simple**- Using safe string handling functions keeps the code simple and helps to minimize errors that could cause security vulnerabilities.  **1. Validate Input Data**- Managing strings effectively requires validating and limiting input size to avoid any buffer overflow risks.  **8. Practice Defense in Depth**- String validation adds an extra layer of defense against threats like buffer overflow and injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | String Handling | Searches for potential unsafe string operations and buffer overflow. |
| SonarQube | 9.3 | C | Looks for insecure string handling that may cause overflows. |
| Coverity | 2024.1 | CERT-STR31-C | Checks for common manipulation issues with strings. |
| Clang Static Analyzer | 12.0.1 | BufferOverflow | Looks for possible buffer overflow conditions that are caused by improper string handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Reduces the risks from unvalidated input in SQL queries, which could enable attackers to exploit and manipulate the database. |

| **Noncompliant Code** |
| --- |
| This example demonstrates insecure code that directly incorporates unchecked user input into SQL queries. |
| std::string userInput = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  executeQuery(query); // This code is susceptible to SQL injection attacks. |

| **Compliant Code** |
| --- |
| The code utilizes prepared statements to safeguard against SQL injection attacks. |
| std::string userInput = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement stmt = connection.prepareStatement(query);  stmt.setString(1, userInput);  executeQuery(stmt); // This approach is secure against SQL injection attacks. |

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

| **Principles(s):**  **6. Adhere to the Principle of Least Privilege**- Restricting how inputs interact with SQL reduces the privileges needed to access the database which in turn limits the potential for abuse.  **1. Validate Input Data**- Utilizing prepared statements ensures that inputs are treated as parameters instead of executable code, this helps to mitigate SQL injection risks.  **8. Practice Defense in Depth**- Prepared statements and input validation act as an added defense to protect against SQL injection even if other security measures fail. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0.1 | SQL Injection | Searches the code to find any unsafe SQK query constructions and gives information on how to correct it. |
| Fortify Static Code Analyzer | 20.2 | SQL Injection | Analyzes code to find vulnerabilities in the SQL Injection and give feedback on secure coding practices. |
| SonarQube | 9.3 | C | Finds instances of SQL queries created using user input and not using prepared statements. |
| Coverity | 2024.1 | CERT.SQL33-C | Looks for the right use of queries with parameters to prevent SQL Injections. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Safeguards the system against memory corruption problems, including buffer overflows and dangling pointers. |

| **Noncompliant Code** |
| --- |
| In this example, memory is allocated without being properly deallocated, resulting in a memory leak. |
| int\* ptr = new int[105]; // Memory is allocated but not released. |

| **Compliant Code** |
| --- |
| Memory is correctly allocated and subsequently released, preventing any memory leaks. |
| int\* ptr = new int[105];  delete[] ptr; // Memory is correctly freed. |

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

| **Principles(s):**  **2. Heed Compiler Warnings-**Compilers often give alerts about memory issues, like leaks. Fixing these promotes effective management of memory.  **9. Use Effective Quality Assurance Technique-**Using secure memory practices, like releasing allocated memory, adds a layer of protection against memory corruption and other vulnerabilities.  **8. Practice Defense in Depth-**Consistent testing with memory analysis tools helps to identify memory leaks and also helps to ensure that the memory is managed appropriately. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | Memory Leak | Searches code for memory leaks and wrong memory deallocations. |
| AddressSanitizer | 12.0.1 | Memory Bugs | Finds memory leaks, use after free, and buffer overflows at runtime |
| Valgrind | 3.17.0 | MemCheck | Finds buffer overflows, memory leaks, and wrong memory deallocations. |
| Clang Static Analyzer | 12.0.1 | MemoryLeak | Finds potential memory leaks in dynamically allocated memory. |
| Coverity | 2024.1 | CERT-MEM31-C | Checks for correct deallocations and allocations of memory in order to prevent dangling pointers and memory leaks. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Validates code assumptions at runtime to detect possible bugs or logic errors early on. |

| **Noncompliant Code** |
| --- |
| In this case, the assertion is used incorrectly, verifying a condition that could be false during typical operation, potentially causing unwanted program termination in a production environment. |
| int divide(int a, int b) {  assert(b != 0); // Program will terminate if b equals 0.  return a / b;  } |

| **Compliant Code** |
| --- |
| Assertions are intended for debugging, not for managing runtime errors. In this case, an appropriate error-handling approach is applied instead. |
| int divide(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Division by zero"); // Correct error handling.  }  return a / b;  } |

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

| **Principles(s):**  **4. Keep It Simple-** Use assertions only for debugging purposes to maintain straightforward code logic, avoiding added complexity in runtime error handling.  **7. Sanitize Data Sent to Other Systems-** Validate inputs to make sure only expected data is sent to prevent harmful or wrong information affecting the system.  **9. Use Effective Quality Assurance Techniques-** Assertions are valuable for debugging and quality control helping detect logical errors made during development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | Assertion Handling | Searches the code for assertions used in production and gives suggestions for correct error handling. |
| Coverity | 2024.1 | CERT-ERR33-C | Looks for wrong use of assertions and ensures correct error handling mechanisms are in place. |
| Clang Static Analyzer | 12.0.1 | Assert Usage | Finds misuse of assertions for runtime error handling. |
| SonarQube | 9.3 | C | Identifies wrong use of assertions for runtime condition checks. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Manages exceptional situations in the code smoothly, preventing crashes or instability in the program. |

| **Noncompliant Code** |
| --- |
| The exception is caught but not adequately handled, which may result in program termination or unpredictable behavior. |
| try {  int result = divide(10, 0);  }  catch (...) {  // Catches all exceptions but does not handle them properly  std::cerr << "An error occurred" << std::endl;  // The program proceeds in an unstable state  } |

| **Compliant Code** |
| --- |
| Exceptions are caught and properly handled, with specific error-handling logic for various exception types. |
| try {  int result = divide(10, 0);  }  catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  // Handles the specific invalid\_argument exception  }  catch (const std::exception& e) {  std::cerr << "General error: " << e.what() << std::endl;  // Handles other general exceptions  } |

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

| **Principles(s):**  **4. Keep It Simple-**Properly managing exceptions keeps the code clean and maintainable ensuring errors are handled in a consistent manner  **8. Practice Defense in Depth-** Effective exception handling adds an extra layer of security, reducing the risk of unexpected crashes or unstable states.  **9. Use Effective Quality Assurance Techniques-** Testing how exceptions are handled during quality assurance ensures that the application responds predictably to errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | Exception Safety | Analyzes the code for correct exception handling and suggests ways to improve them. |
| Coverity | 2024.1 | CERT-ERR09-C | Looks for proper use of exception handling and ensuring that all exceptions that are caught are managed correctly. |
| Clang Static Analyzer | 12.0.1 | Exception Handling | Finds potential misuse of exceptions and error handling patterns. |
| SonarQube | 9.3 | C | Finds where exceptions are caught but not handled correctly. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Validation** | [STD-008-CPP] | Makes sure that all system inputs are thoroughly validated before processing, helping to mitigate risks such as buffer overflows, command injection, and SQL injection. |

| **Noncompliant Code** |
| --- |
| In this example, the user input is not properly validated, permitting unsafe data to enter the system and potentially causing a buffer overflow or other security issues. |
| char buffer[10];  std::cin >> buffer; // No validation of input size, which could cause a buffer overflow. |

| **Compliant Code** |
| --- |
| The input is validated to ensure the buffer size is not exceeded, effectively preventing potential buffer overflow. |
| char buffer[10];  std::cin.get(buffer, sizeof(buffer)); // Restricts input to prevent buffer overflow. |

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

| **Principles(s):**  **1. Validate Input Data-**This focuses on verifying that all incoming data is properly checked and constrained to safe values to prevent overflows and other risks.  **8. Practice Defense in Depth-** Input validation acts as a vital layer of defense against threats like buffer overflows and injections.  **9. Use Effective Quality Assurance Techniques-** Thoroughly testing input validation mechanisms ensures vulnerabilities are found and resolved before they become exploitable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | Input Handling | Searches the code for input validation errors. |
| Coverity | 2024.1 | CERT-STR50-C | Looks for proper validation of input data to prevent buffer overflow and any other input vulnerabilities. |
| SonarQube | 9.3 | C | Finds cases where input is not correctly validated which increases the risk of buffer overflow. |
| Clang Static Analyzer | 12.0.1 | Input Validation | Finds potential input validation issues like unchecked buffer sizes. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | [STD-009-CPP] | Ensures that errors are managed securely and consistently, avoiding unintended behavior or the release of sensitive data. |

| **Noncompliant Code** |
| --- |
| Errors are not managed, allowing the program to proceed in an unstable state, which may result in crashes or unpredictable behavior. |
| int divide(int a, int b) {  return a / b; // Does not have a check for division by zero.  } |

| **Compliant Code** |
| --- |
| Errors are effectively handled, ensuring that the program operates as expected even in exceptional situations. |
| int divide(int a, int b) {  if (b == 0) {  std::cerr << "Error: Division by zero" << std::endl;  return 0; // Handles the error and prevents a crash.  }  return a / b;  } |

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

| **Principles(s):**  **4. Keep it Simple-** Effective error handling keeps the code straightforward and predictable which minimizes undefined behavior and simplifies the debugging and maintenance.  **8. Practice Defense in Depth-** Error handling serves as a key layer of defense which helps to prevent application crashes and unexpected behaviors that could lead to security problems.  **9. Use Effective Quality Assurance Techniques-** Testing how the application handles errors ensures secure and reliable response which in turn contributes to a strong quality assurance process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | Error Management | Searches the code for any potential errors in error handling and give feedback on ways to correct it to prevent crashes. |
| Coverity | 2024.1 | CERT-ERR08-C | Loos for comprehensive error handling to ensure that all of the error cases are securely managed. |
| SonarQube | 9.3 | C | Finds instances where error conditions are not managed correctly and gives feedback on correct and safe practices for error handling. |
| Clang Static Analyzer | 12.0.1 | Error Handling | Finds cases of improper or missing error handling. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | [STD-010-CPP] | Effectively manages memory and system resources to avoid memory leaks, resource depletion, and associated security risks. |

| **Noncompliant Code** |
| --- |
| This example demonstrates poor resource management, where a file is opened but not properly closed, causing a resource leak. |
| std::fstream file("example.txt", std::ios::in); // File remains open which leads to a resource leak. |

| **Compliant Code** |
| --- |
| In this case, the file is managed correctly using RAII (Resource Acquisition Is Initialization), ensuring that it is automatically closed when the resource goes out of scope. |
| {  std::fstream file("example.txt", std::ios::in);  // The file is automatically closed when the scope ends.  } // Resource is released. |

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

| **Principles(s):**  **2. Heed Compiler Warnings-** Compilers frequently warn about resource leaks. Resolving these warnings promotes efficient resource management.  **8. Practice Defense in Depth-** Managing resources properly adds a defensive layer, ensuring they are released and preventing issues like memory leaks and exhaustion.  **9. Use Effective Quality Assurance Techniques-** Regular testing, including static analysis, helps identify resource mismanagement and ensures efficient usage without leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | 12.0.1 | Resource Use | Finds resource leaks and misuse during the execution of the program |
| Cppcheck | 2.8 | Resource Management | Searches the code for possible resource leaks. |
| Coverity | 2024.1 | CERT-STR31-C | Looks for correct resource management to ensure that resources are released correctly. |
| Valgrind | 3.17.0 | Memcheck | Finds improper memory usage, mismanagement of resources, and memory leaks at runtime. |
| Clang Static Analyzer | 12.0.1 | Resource Leak | Finds potential resource leaks like file handles that are unclosed. |

### 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.

Automation is essential for enforcing the standards detailed in this security policy. By integrating security tools and practices within Green Pace’s DevOps workflow, automation can streamline vulnerability assessments, code analysis, and compliance checks throughout the development lifecycle.

Based on the DevSecOps model, automation should be incorporated during the "Verify and Test" phase in pre-production and continue into production. This involves:

* **Design Phase**: Embedding security testing as part of the design requirements to ensure secure coding practices are integrated from the outset.
* **Build Phase**: Employing static analysis tools such as SonarQube and Clang Static Analyzer to detect code vulnerabilities and compliance issues during the build process.
* **Verify and Test Phase**: Automating tests to detect issues such as memory leaks, input validation errors, and resource mismanagement before deployment.
* **Monitor and Detect Phase**: Using automated monitoring solutions to continuously evaluate the production environment for compliance and identify potential breaches or deviations from coding standards.

By embedding automation into the DevSecOps pipeline, Green Pace can enforce coding standards, identify vulnerabilities early, and maintain consistent compliance throughout development.

### 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 | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | High | 1 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-006-CPP | Medium | Likely | Low | Medium | 2 |
| STD-007-CPP | High | Likely | Medium | High | 1 |
| STD-008-CPP | High | Likely | Medium | High | 1 |
| STD-009-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-010-CPP | High | Likely | Medium | High | 1 |

### 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 | **Explanation-** Encryption at rest involves securing data stored on disk. This policy aims to protect sensitive information, even in the event of a storage compromise. It covers all customer data, internal records, and sensitive information held on Green Pace's servers or databases.  **Application of the Policy-** This policy is applicable during data storage and backup activities. Data must be encrypted with robust encryption algorithms before being saved to storage devices. |
| Encryption in flight | **Explanation-** Encryption in transit secures data while it moves across networks. It utilizes protocols such as TLS/SSL to ensure data transmitted between clients, servers, and APIs is protected, preventing unauthorized interception or eavesdropping.  **Application of the Policy-** This policy is applicable whenever data is transmitted between internal systems, customer applications, or external networks. |
| Encryption in use | **Explanation-** Encryption in use provides security for data while it is actively processed in memory. Techniques such as hardware-based memory encryption are employed to protect sensitive information during processing.  **Application of the Policy-** This policy applies when dealing with sensitive data in applications requiring real-time processing, including financial transactions or personal data analysis. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Explanation-** Authentication involves confirming the identity of users and systems, ensuring that only authorized individuals can access resources, applications, and data. This policy mandates the use of strong, multi-factor authentication methods for all users.  **Application of the Policy-** This policy is applicable whenever accessing systems, networks, applications, or data. |
| Authorization | **Explanation-** Authorization defines the actions that authenticated users can perform within a system. This policy ensures that users are granted access only to the resources and information required for their roles, following the principle of least privilege.  **Application of the Policy-** This policy applies during access control verification for resources, data, and application features. |
| Accounting | **Explanation-** Accounting refers to tracking user actions within a system to create an audit trail. This policy ensures that all access, changes, and interactions involving sensitive information are logged for review and analysis.  **Application of the Policy-** This policy is always in effect, using logging and monitoring tools to record activities like login attempts, data access, and system changes. |

**\***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 | 10/11/2024 | Complete Project One | Robert Cook |  |
| [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 |