**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 means making sure that the stuff people put into a software system makes sense and won't cause problems. It's like checking that the information fits the rules—it's the right type, size, and format. This helps keep things running smoothly and stops bad stuff like hacks or crashes. When input is checked well, it makes the software more reliable, secure, and easier to use because it catches mistakes and gives clear feedback when things aren't right. |
| 1. Heed Compiler Warnings | Header compiler warnings are like helpful reminders from your computer when you're writing code. They pop up at the top of your screen to let you know about things that might be wrong with your code, like typos or things that could cause problems later on. They're there to help you fix stuff before it becomes a big headache, so paying attention to them can save you time and make sure your code works smoothly. |
| 1. Architect and Design for Security Policies | Architecting and designing for security policies means planning your software so it keeps things safe without being a hassle. It's like setting up your house with good locks and alarms to keep out bad guys. You think about where the windows are and how to keep everything secure without making it hard for you to get in and out. This way, your software stays safe from hackers and other troublemakers, and you can relax knowing it's well protected. |
| 1. Keep It Simple | Keeping it simple in security means not making things more complicated than they need to be. It's like using a straightforward lock that does the job without lots of fancy features that might confuse you or others. By keeping things simple, you make it easier to understand what's going on and easier to spot any potential problems. This way, you can focus on keeping things safe without adding unnecessary complexity. |
| 1. Default Deny | Default deny is like saying 'no' to everything unless you have a good reason to say 'yes.' It's like keeping your front door locked unless you know and trust the person knocking. This principle keeps things safe by blocking access by default and only allowing what's necessary, which helps prevent unwanted surprises or troublemakers from getting in. It's a simple way to stay in control and protect what's important. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the Principle of Least Privilege is like giving people just enough access to do their job, but not more than they need. It's like handing out keys—you only give someone the key to the room they need to be in, not the whole building. This way, if something goes wrong, the damage is limited because everyone only has access to what they absolutely need. It's a smart way to keep things safe and make sure no one messes with stuff they shouldn't. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems is like cleaning up before you send a package to someone. It's making sure there's nothing weird or harmful in the data you're sharing, like bad words or sneaky codes that could cause problems on the other end. By checking and cleaning the data, you make sure it's safe to send and won't mess up the system it's going to. It's a bit like double-checking your gift wrap to make sure it's neat and tidy before sending it off. |
| 1. Practice Defense in Depth | Practicing defense in depth is like having multiple layers of security around your stuff. It's not just relying on one lock on your door—it's also having an alarm, maybe a guard dog, and a neighborhood watch. Each layer adds extra protection so even if one fails, there are others to keep things safe. It's like having backup plans for your backup plans, making it harder for bad stuff to get through and causing less worry about what might happen. |
| 1. Use Effective Quality Assurance Techniques | Using effective quality assurance techniques is like checking your work before handing it in. It's about making sure everything works like it should and there aren't any sneaky mistakes hiding in there. It's a bit like tasting the soup before serving it—making sure it's just right and won't leave anyone with a bad taste. By double-checking everything, you can catch problems early and make sure your software runs smoothly without surprises. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard is like following a recipe for baking a cake. It's about using the right ingredients in the right amounts to make sure the cake turns out tasty and safe to eat. In coding, it means using rules and guidelines to write software that's less likely to have bugs or be vulnerable to hackers. It's a bit like having a checklist to make sure you don't forget any important steps—keeping things organized and reliable from start to finish. |

### 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-DAT] | Variable Declaration and Initialization - Proper variable declaration and initialization ensure variables are defined with a specific type and are initialized before use to avoid undefined behavior and improve code readability and maintainability. |

| **Noncompliant Code** |
| --- |
| Variables are declared without being initialized, leading to undefined behavior when the variables are used before assignment. |
| int main() {  int x; // Variable declared but not initialized  printf("%d\n", x); // Undefined behavior, x is uninitialized  return 0;  }  void function() {  float y; // Variable declared but not initialized  printf("%f\n", y); // Undefined behavior, y is uninitialized  } |

| **Compliant Code** |
| --- |
| Variables are declared and initialized at the time of declaration, ensuring defined behavior and improving code safety. |
| int main() {  int x = 0; // Variable declared and initialized  printf("%d\n", x); // Defined behavior, prints 0  return 0;  }  void function() {  float y = 0.0; // Variable declared and initialized  printf("%f\n", y); // Defined behavior, prints 0.000000  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | UninitializedVariableCheck | Checks for uninitialized variables |
| Coverity | 2021.12 | CWE-457 | Identifies instances of uninitialized variables |
| CodeQL | 2.5.7 | Cpp/uninitialized-local | Detects local variables that are uninitialized. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-VAL] | Ensuring Valid Data Values - Ensuring variables hold valid and expected data values prevents logical errors and enhances the robustness and correctness of the program. It helps avoid unexpected behaviors, especially in conditional checks and calculations. |

| **Noncompliant Code** |
| --- |
| The variable is assigned a value outside the valid range, which can lead to incorrect behavior in the program. |
| #include <stdio.h>  int main() {  int age = -5; // Invalid age value  if (age >= 0 && age <= 120) {  printf("Valid age\n");  } else {  printf("Invalid age\n");  }  return 0;  }  #include <stdio.h>  void checkScore() {  int score = 105; // Invalid score value  if (score >= 0 && score <= 100) {  printf("Valid score\n");  } else {  printf("Invalid score\n");  }  } |

| **Compliant Code** |
| --- |
| The variable is assigned a value within the valid range, ensuring the program behaves correctly. |
| #include <stdio.h>  int main() {  int age = 25; // Valid age value  if (age >= 0 && age <= 120) {  printf("Valid age\n");  } else {  printf("Invalid age\n");  }  return 0;  }  #include <stdio.h>  void checkScore() {  int score = 85; // Valid score value  if (score >= 0 && score <= 100) {  printf("Valid score\n");  } else {  printf("Invalid score\n");  }  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | DataValueCheck | Checks for invalid data values |
| Coverity | 2021.12 | MISRA-C-2012-Rule-10.1 | Ensures correct data value assignment |
| CodeQL | 2.5.7 | Cpp/invalid-value | Detects invalid values assigned to variables |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | Ensuring String Correctness - Proper handling of strings is essential to avoid security vulnerabilities such as buffer overflows and to ensure the correct functioning of the program. This includes ensuring strings are properly null-terminated, allocated with sufficient space, and manipulated safely. |

| **Noncompliant Code** |
| --- |
| The string is not properly null-terminated, leading to undefined behavior when the string is used. |
| #include <stdio.h>  #include <string.h>  int main() {  char str[5];  strcpy(str, "hello"); // Buffer overflow, str is not large enough  printf("%s\n", str); // Undefined behavior  return 0;  }  #include <stdio.h>  #include <string.h>  int main() {  char buffer[10];  strcpy(buffer, "This is too long"); // Buffer overflow, buffer is not large enough  printf("%s\n", buffer); // Undefined behavior  return 0;  } |

| **Compliant Code** |
| --- |
| The string is properly null-terminated and the buffer is adequately sized to prevent overflow. |
| #include <stdio.h>  #include <string.h>  int main() {  char str[6]; // Allocate enough space for "hello" and null terminator  strncpy(str, "hello", sizeof(str) - 1);  str[sizeof(str) - 1] = '\0'; // Ensure null termination  printf("%s\n", str); // Defined behavior, prints "hello"  return 0;  }  #include <stdio.h>  #include <string.h>  int main() {  char buffer[20]; // Allocate enough space  strncpy(buffer, "This is safe", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination  printf("%s\n", buffer); // Defined behavior, prints "This is safe"  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.1 | Buffer Overflow Checker | Analyzes code to detect potential buffer overflow issues and ensures proper string handling practices are followed. |
| SonarQube | 9.0 | Security Vulnerability Detection | Scans code for security vulnerabilities related to string manipulation and provides recommendations for improving string handling. |
| Clang Static Analyzer | 15.0 | Undefined Behavior Detector | Identifies instances of undefined behavior related to string operations, helping developers correct improper string handling. |
| RIPS Code Analysis | 3.5 | Security Code Analysis | Detects security issues in source code related to string handling and buffer overflows, ensuring compliance with secure coding standards. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | Preventing SQL Injection - SQL injection is a serious security vulnerability that can allow attackers to execute arbitrary SQL code on a database, leading to data breaches and other malicious activities. Properly handling user inputs and using parameterized queries can prevent SQL injection. |

| **Noncompliant Code** |
| --- |
| The code directly incorporates user input into an SQL query without validation or parameterization, making it vulnerable to SQL injection. |
| #include <stdio.h>  #include <stdlib.h>  #include <sqlite3.h>  int main() {  sqlite3 \*db;  sqlite3\_open("example.db", &db);    char username[50];  printf("Enter username: ");  scanf("%s", username);    char sql[100];  sprintf(sql, "SELECT \* FROM users WHERE username = '%s';", username); // Vulnerable to SQL injection    sqlite3\_exec(db, sql, 0, 0, 0);  sqlite3\_close(db);  return 0;  }  #include <stdio.h>  #include <stdlib.h>  #include <sqlite3.h>  int main() {  sqlite3 \*db;  sqlite3\_open("example.db", &db);  char username[50];  printf("Enter username: ");  scanf("%49s", username);  char sql[100];  snprintf(sql, sizeof(sql), "DELETE FROM users WHERE username='%s';", username); // Vulnerable to SQL injection  sqlite3\_exec(db, sql, 0, 0, 0);  sqlite3\_close(db);  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses a parameterized query to safely incorporate user input, preventing SQL injection. |
| #include <stdio.h>  #include <stdlib.h>  #include <sqlite3.h>  int main() {  sqlite3 \*db;  sqlite3\_open("example.db", &db);    char username[50];  printf("Enter username: ");  scanf("%s", username);    const char \*sql = "SELECT \* FROM users WHERE username = ?;";  sqlite3\_stmt \*stmt;  sqlite3\_prepare\_v2(db, sql, -1, &stmt, 0);  sqlite3\_bind\_text(stmt, 1, username, -1, SQLITE\_STATIC);    while (sqlite3\_step(stmt) == SQLITE\_ROW) {  printf("User found: %s\n", sqlite3\_column\_text(stmt, 0));  }    sqlite3\_finalize(stmt);  sqlite3\_close(db);  return 0;  }  #include <stdio.h>  #include <stdlib.h>  #include <sqlite3.h>  int main() {  sqlite3 \*db;  sqlite3\_open("example.db", &db);  char username[50];  printf("Enter username: ");  scanf("%49s", username);  const char \*sql = "DELETE FROM users WHERE username = ?;";  sqlite3\_stmt \*stmt;  sqlite3\_prepare\_v2(db, sql, -1, &stmt, 0);  sqlite3\_bind\_text(stmt, 1, username, -1, SQLITE\_STATIC);  sqlite3\_step(stmt);  sqlite3\_finalize(stmt);  sqlite3\_close(db);  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  8. Practice Defense in Depth - Practicing defense in depth is like having multiple layers of security around your stuff. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.12.0 | SQL Injection Scanner | An open-source security scanner that detects SQL injection vulnerabilities in web applications by analyzing various input points. |
| SQLMap | 1.6.5 | SQL Injection Detection | An automated tool that identifies and exploits SQL injection flaws, providing detailed reports and remediation suggestions. |
| Veracode | 2024.1 | Static Analysis Security Testing (SAST) | A static code analysis tool that detects SQL injection vulnerabilities and other security issues in source code. |
| Fortify | 2024.1 | Security Code Analyzer | Provides comprehensive analysis to detect SQL injection vulnerabilities in code and offers recommendations for remediation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | Ensuring Memory Protection - Proper memory management is crucial for the stability and security of programs. It prevents common vulnerabilities such as memory leaks, buffer overflows, and use-after-free errors, which can lead to program crashes and security breaches. |

| **Noncompliant Code** |
| --- |
| The code allocates memory but fails to deallocate it, leading to a memory leak. |
| #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr = (int \*)malloc(sizeof(int) \* 10); // Allocate memory  if (ptr == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  // Use the allocated memory  for (int i = 0; i < 10; i++) {  ptr[i] = i;  }  // Memory is not freed  return 0;  }  #include <stdio.h>  #include <stdlib.h>  int main() {  char \*buffer = (char \*)malloc(256); // Allocate memory  if (buffer == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  // Use the allocated memory  snprintf(buffer, 256, "Hello, World!");  // Forgot to free the allocated memory  return 0;  } |

| **Compliant Code** |
| --- |
| The code allocates memory and ensures it is properly deallocated, preventing memory leaks. |
| #include <stdio.h>  #include <stdlib.h>  int main() {  int \*ptr = (int \*)malloc(sizeof(int) \* 10); // Allocate memory  if (ptr == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  // Use the allocated memory  for (int i = 0; i < 10; i++) {  ptr[i] = i;  }  free(ptr); // Free the allocated memory  return 0;  }  #include <stdio.h>  #include <stdlib.h>  int main() {  char \*buffer = (char \*)malloc(256); // Allocate memory  if (buffer == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  // Use the allocated memory  snprintf(buffer, 256, "Hello, World!");  free(buffer); // Free the allocated memory  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  8. Practice Defense in Depth - Practicing defense in depth is like having multiple layers of security around your stuff. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20.0 | Memory Leak Detector | An open-source tool that detects memory leaks, memory corruption, and other memory-related issues in programs. |
| AddressSanitizer | LLVM 15.0 | Memory Error Detector | A runtime memory error detector that helps identify memory leaks, buffer overflows, and use-after-free errors in code. |
| Dr. Memory | 2.2.0 | Memory Leak and Error Detection | A tool for detecting memory leaks and other memory-related programming errors during runtime. |
| Purify | 2024.1 | Memory Management Checker | A commercial tool that analyzes memory usage to find leaks, overflows, and other memory-related issues. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASR] | Using Assertions - Assertions are used to enforce program invariants and catch programming errors early during development. They help ensure that the program behaves as expected and can prevent subtle bugs from causing issues later in the development cycle. |

| **Noncompliant Code** |
| --- |
| The code does not use assertions to validate critical assumptions, which can lead to undetected errors and unexpected behavior. |
| #include <stdio.h>  int divide(int a, int b) {  // No assertion to check if 'b' is zero  return a / b;  }  int main() {  int result = divide(10, 0); // This will cause a runtime error  printf("Result: %d\n", result);  return 0;  }  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*array = (int \*)calloc(20, sizeof(int)); // Allocate memory  if (array == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  // Use the allocated memory  for (int i = 0; i < 20; i++) {  array[i] = i \* 2;  }  // Forgot to free the allocated memory  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses assertions to validate critical assumptions, ensuring that errors are caught early and handled appropriately. |
| #include <stdio.h>  #include <assert.h>  int divide(int a, int b) {  assert(b != 0); // Assert that 'b' is not zero  return a / b;  }  int main() {  int result = divide(10, 0); // Assertion will catch the invalid input  printf("Result: %d\n", result);  return 0;  }  #include <stdio.h>  #include <stdlib.h>  int main() {  int \*array = (int \*)calloc(20, sizeof(int)); // Allocate memory  if (array == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  // Use the allocated memory  for (int i = 0; i < 20; i++) {  array[i] = i \* 2;  }  free(array); // Free the allocated memory  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  2. Keep It Simple |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Analysis Tools | 2024.1 | Assertion Usage Analyzer | Analyzes code to check for the presence and correctness of assertions, ensuring that critical assumptions are validated. |
| Clang Static Analyzer | 15.0 | Assertion Detection | Detects missing assertions and verifies that assertions are correctly used to enforce program invariants. |
| SonarQube | 9.0 | Code Quality Analysis | Reviews code for the use of assertions and provides recommendations for improving code reliability by ensuring that important conditions are asserted. |
| Coverity | 2024.1 | Software Defect Analyzer | Identifies areas in code where assertions could be beneficial for catching programming errors and improving program correctness. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXP] | Proper Handling of Exceptions - Proper handling of exceptions is crucial to ensure that programs can gracefully handle unexpected situations and errors. This prevents crashes and undefined behavior, providing a better user experience and more reliable software. |

| **Noncompliant Code** |
| --- |
| The code does not properly handle exceptions, leading to potential crashes and undefined behavior when an error occurs. |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  throw std::runtime\_error("An error occurred");  }  int main() {  riskyOperation(); // No exception handling  std::cout << "This will not be printed if an exception occurs\n";  return 0;  }  #include <iostream>  #include <stdexcept>  void openFile() {  throw std::runtime\_error("Failed to open file");  }  int main() {  openFile(); // No exception handling  std::cout << "This will not be printed if an exception occurs\n";  return 0;  } |

| **Compliant Code** |
| --- |
| The code uses try-catch blocks to properly handle exceptions, ensuring that errors are caught and managed gracefully. |
| #include <iostream>  #include <stdexcept>  void riskyOperation() {  throw std::runtime\_error("An error occurred");  }  int main() {  try {  riskyOperation();  } catch (const std::exception &e) {  std::cerr << "Exception caught: " << e.what() << '\n';  // Handle the error appropriately  }  std::cout << "This will be printed even if an exception occurs\n";  return 0;  }  #include <iostream>  #include <stdexcept>  void openFile() {  throw std::runtime\_error("Failed to open file");  }  int main() {  try {  openFile();  } catch (const std::runtime\_error &e) {  std::cerr << "Exception caught: " << e.what() << '\n';  // Handle the error appropriately  }  std::cout << "This will be printed even if an exception occurs\n";  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0 | Exception Handling Checker | Analyzes code to ensure that exceptions are properly handled and that try-catch blocks are used appropriately. |
| SonarQube | 9.0 | Exception Handling Rules | Reviews code for proper exception handling practices and provides recommendations for improving error management. |
| Coverity | 2024.1 | Exception Handling Analysis | Identifies missing or incorrect exception handling in code, helping developers ensure that all exceptions are properly caught and managed. |
| Visual Studio Code Analysis | 2024.1 | Exception Handling Analyzer | Evaluates code to check for proper usage of exception handling mechanisms and offers suggestions for handling errors more effectively. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Code Documentation | [STD-008-DOC] | Code Documentation - Proper documentation is critical for maintaining code quality, aiding in future development, and ensuring that other developers can understand and effectively work with the code. Clear and concise comments and documentation improve code readability and maintainability. |

| **Noncompliant Code** |
| --- |
| The code lacks comments and documentation, making it difficult to understand and maintain. |
| #include <stdio.h>  int add(int a, int b) {  return a + b;  }  int main() {  int result = add(5, 3);  printf("Result: %d\n", result);  return 0;  }  #include <iostream>  void processArray(int \*arr, int size) {  if (size < 0) {  throw std::invalid\_argument("Array size cannot be negative");  }  // Further processing  }  int main() {  int \*arr = nullptr;  processArray(arr, -1); // No exception handling  std::cout << "This will not be printed if an exception occurs\n";  return 0;  } |

| **Compliant Code** |
| --- |
| The code includes clear and concise comments and documentation, making it easier to understand and maintain. |
| #include <stdio.h>  /\*\*  \* @brief Adds two integers.  \*  \* This function takes two integers as input and returns their sum.  \*  \* @param a First integer to add.  \* @param b Second integer to add.  \* @return Sum of a and b.  \*/  int add(int a, int b) {  return a + b;  }  /\*\*  \* @brief Main function.  \*  \* This is the main entry point of the program. It calls the add function  \* and prints the result.  \*  \* @return Returns 0 on success.  \*/  int main() {  int result = add(5, 3);  printf("Result: %d\n", result);  return 0;  }  #include <iostream>  #include <stdexcept>  void processArray(int \*arr, int size) {  if (size < 0) {  throw std::invalid\_argument("Array size cannot be negative");  }  // Further processing  }  int main() {  try {  int \*arr = nullptr;  processArray(arr, -1);  } catch (const std::invalid\_argument &e) {  std::cerr << "Exception caught: " << e.what() << '\n';  // Handle the error appropriately  }  std::cout << "This will be printed even if an exception occurs\n";  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Doxygen | 1.9.4 | Documentation Generator | A tool that generates comprehensive documentation from annotated source code comments, ensuring that code is well-documented and easy to understand. |
| Javadoc | 1.8.0 | Java Documentation Generator | A tool for generating API documentation in HTML format from comments in Java source code, improving code clarity and maintainability. |
| Sphinx | 5.3.0 | Documentation Builder | A documentation generator that creates readable documentation from reStructuredText sources, often used with Python code. |
| DocFX | 2.0.6 | Documentation Generator | A tool for generating static documentation from source code and markdown files, enhancing code readability and maintainability. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | [STD-009-RES] | Proper Resource Management - Proper resource management is critical to ensure that system resources such as file handles, network connections, and memory are properly acquired and released. This prevents resource leaks, ensures system stability, and maintains the overall performance of the application. |

| **Noncompliant Code** |
| --- |
| The code opens a file but does not close it, leading to a resource leak. |
| #include <stdio.h>  int main() {  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  return 1;  }  // Read from the file (omitted for brevity)    // The file is not closed, leading to a resource leak  return 0;  }  #include <stdio.h>  int main() {  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  return 1;  }  // Read from the file (omitted for brevity)  // The file is not closed, leading to a resource leak  return 0;  } |

| **Compliant Code** |
| --- |
| The code properly opens and closes the file, ensuring that the resource is released. |
| #include <stdio.h>  int main() {  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  return 1;  }  // Read from the file (omitted for brevity)  fclose(file); // Properly close the file to release the resource  return 0;  }  #include <stdio.h>  int main() {  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  return 1;  }  // Read from the file (omitted for brevity)    fclose(file); // Properly close the file to release the resource  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20.0 | Resource Leak Detector | An open-source tool that detects resource leaks, such as file handles and memory, helping ensure that resources are properly managed and released. |
| AddressSanitizer | LLVM 15.0 | Resource Leak Detector | A runtime memory error detector that helps identify resource leaks, including file and network resources, in code. |
| Dr. Memory | 2.2.0 | Resource Leak Analysis | A tool for detecting resource leaks and other resource-related issues during runtime, ensuring proper management of system resources. |
| LeakSanitizer | 2024.1 | Resource Leak Detector | Part of the AddressSanitizer suite, LeakSanitizer focuses on detecting memory and resource leaks in code to ensure efficient resource management. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Proper Error Handling and Recovery. | [STD-010-ERR] | Proper Error Handling and Recovery - Proper error handling is crucial to ensure that software can gracefully recover from unexpected conditions or errors. This enhances reliability, prevents crashes, and provides meaningful feedback to users or administrators. |

| **Noncompliant Code** |
| --- |
| The code does not handle errors or unexpected conditions, potentially leading to crashes or undefined behavior. |
| #include <stdio.h>  int main() {  FILE \*file = fopen("nonexistent.txt", "r");  // Attempt to read from the file without checking if it opened successfully  char buffer[256];  fread(buffer, sizeof(char), sizeof(buffer), file);  fclose(file);  return 0;  }  #include <stdio.h>  #include <stdlib.h>  int main() {  char \*buffer = (char \*)malloc(256);  if (buffer == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  free(buffer); // Memory is freed, but file is not closed  return 1;  }  // Read from the file (omitted for brevity)    // File is not closed, leading to a resource leak  free(buffer); // Memory is freed  return 0;  } |

| **Compliant Code** |
| --- |
| The code checks for errors when opening the file and handles them gracefully. It also ensures proper resource management by closing the file even if an error occurs. |
| #include <stdio.h>  int main() {  FILE \*file = fopen("nonexistent.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  return 1;  }  // Read from the file (omitted for brevity)  fclose(file); // Properly close the file to release the resource  return 0;  }  #include <stdio.h>  #include <stdlib.h>  int main() {  char \*buffer = (char \*)malloc(256);  if (buffer == NULL) {  fprintf(stderr, "Memory allocation failed\n");  return 1;  }  FILE \*file = fopen("example.txt", "r");  if (file == NULL) {  fprintf(stderr, "Error opening file\n");  free(buffer); // Memory is freed  return 1;  }  // Read from the file (omitted for brevity)    fclose(file); // Properly close the file to release the resource  free(buffer); // Properly free the allocated memory  return 0;  } |

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

| **Principles(s):** 1. Validate Input Data - Make sure that the stuff people put into a software system makes sense and won't cause problems.  4. Keep It Simple - Keeping it simple in security means not making things more complicated than they need to be. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20.0 | Error Detection | Detects various runtime errors, including issues related to file operations and resource management, helping identify areas where error handling may be lacking. |
| Static Analysis Tools (e.g., Clang Static Analyzer) | 15.0 | Error Handling Analysis | Analyzes code for proper error handling practices, identifying potential issues where errors are not checked or handled appropriately. |
| Coverity | 2024.1 | Error Handling and Resource Management | Provides analysis to ensure that errors are properly handled and resources are managed correctly, helping to identify and fix error handling deficiencies. |
| SonarQube | 9.0 | Error Handling Rules | Reviews code for adherence to error handling best practices, offering recommendations for improving error detection and recovery. |

### 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

To integrate automation for enforcing and ensuring compliance with our coding standards, we can embed security checks throughout our DevSecOps pipeline.

In the Assess and Plan phase, automated tools can monitor threats and regulatory changes, prioritizing updates. During Design, static analysis tools can enforce best practices in real-time. In the Build phase, CI/CD pipelines can include security checks, catching vulnerabilities early.

For Verify and Test, automated security and compliance testing ensures secure behavior. Transition and Health Check use automated scripts for consistent security settings and routine penetration tests.

In Production, monitoring tools provide real-time alerts for anomalies. Automated response systems quickly mitigate issues by blocking attacks or rolling back services. Finally, Maintain and Stabilize with regular automated assessments ensures ongoing compliance and stability. This approach integrates security seamlessly into our existing DevOps process, enhancing our security posture.

### 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-DAT | Medium | Likely | Low | High | 1 |
| STD-002-VAL | Medium | Medium | Low | Medium | 2 |
| STD-003-STR | High | Medium | Medium | High | 5 |
| STD-004-SQL | Critical | High | Medium | High | 5 |
| STD-005-MEM | High | Medium | Medium | High | 4 |
| STD-006-ASR | Medium | Medium | Low | Medium | 3 |
| STD-007-EXP | High | Medium | Medium | High | 4 |
| STD-008-DOC | Low | Medium | Low | Medium | 2 |
| STD-009-RES | Medium | Medium | Low | Medium | 3 |
| STD-010-ERR | High | Mediu | Medium | High | 4 |

### 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 | Encrypting data stored on disk. Ensures data is secure even if physical access to the storage medium is obtained |
| Encryption in flight | Encrypting data transmitted over the network. Prevents interception and eaves dropping during data transmission. |
| Encryption in use | Encrypting data while it is being processed. Protects sensitive data from being accessed by unauthorized processes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying the identity of users and systems. Ensures only authorized entities can access resources. |
| Authorization | Granting permissions to authenticated users and systems. Ensures users have appropriate access levels |
| Accounting | Recording and analyzing activities. Ensures accountability and aids in detecting security breaches. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | [Insert text.] | Updated Security Protocols | Robert Bogart | [Insert text.] |
| 1.2 | [Insert text.] | Revised encryption standards. | Robert Bogart | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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