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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data can prevent common security concerns such as injection attacks, buffer overflows, etc. Validation can come in the form of ensuring that all input data is confined to set boundaries including data types, ranges, and formats. |
| 1. Heed Compiler Warnings | Compiler warnings are useful ways to start addressing potential security threats. They often indicate errors that could lead to vulnerabilities, and act as a guide for developers. Compiler warnings can also help the programmer improve overall readability, transferability, and reliability of the code. |
| 1. Architect and Design for Security Policies | Security policies can be designed and built into the program, addressing security issues early instead of backtracking after the program is complete. Implementing basic principles like the principle of least privilege and secure data storage from the start of the project can make a big difference in the overall security of the program. |
| 1. Keep It Simple | Keeping the design of the program simple can improve security. Modularizing code ensures readability and reduces complexity that can create unforeseen vulnerabilities. |
| 1. Default Deny | Defaulting to deny access without specific controls or permissions is a simple but effective way to improve security. It can be considered the first line of defense against potential security threats, as the default is to deny all access at first, creating an initial hurdle for malicious users to overcome. |
| 1. Adhere to the Principle of Least Privilege | As part of the Architect and Design for Security Policies, the principle of least privilege means that every user is granted as little privilege in the program as possible. Their access is limited to only the functions and features they need for how they will use the program. Limiting access reduces the risk of unintentional data breaches. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data reduces the risk of vulnerabilities by ensuring that the data transmitted is free from potential exploits. Encryption, content inspection, format standardization for data, and filtration are all methods of sanitizing data. |
| 1. Practice Defense in Depth | Creating a multi-layered approach ensures that malicious users have to overcome several layers of defense before having access to any data. Part of a multi-layered approach is having contingency plans if there is a breach and a response team to address it. Monitoring use and having strict standards for user authentication are other ways to create a multi-layered approach. |
| 1. Use Effective Quality Assurance Techniques | Implementing testing to ensure the program meets certain requirements both in functionality and security is a way to create a secure program. Programs that do not function at a certain level can expose vulnerabilities to malicious users. Creating a continuous program of quality assurance and testing can ensure long-term security and reliability. |
| 1. Adopt a Secure Coding Standard | Using the SEI CERT C++ Coding Standard to address common security vulnerabilities creates a resilient program that is maintainable, reliable, and secure. |

### 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** | **Data Type Coding Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-DTC] | Having a standardized naming and typing conventions ensures the maintainability, transferability, readability and overall performance. It also ensures integrity of the data. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because it is using an ‘int’ variable for the loop index instead of an unsigned integer or a ‘size\_t’ . This can produce unintentional negative values. |
| #include <iostream>  int main() {  int n;  for (n = 0; n < 10; ++n) {  std::cout << "Iteration: " << n << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code uses the loop index of ‘size\_t’ instead of ‘int’, ensuring the clarity, portability, and reliability of the code. |
| #include <iostream>  int main() {  size\_t n; // Using size\_t for loop index  for (n = 0; n < 10; ++n) {  std::cout << "Iteration: " << n << std::endl;  }  return 0;  } |

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

| **Principles(s):**  (1: Validate Input Data: Validating input data ensures that expected formats and types are conformed to, maintaining the integrity of the data, as well as program predictability and stability.)  (4: Keep It Simple: Keeping the design of the program simple by maintaining a standardized naming system reduces complexity. Modularizing the code ensures readability, transferability, and overall stability due to consistency. ) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.3 | enable=warning  style  performance  portability  unusedFunction | analysis tool that is static, it can detect potential issues including input validation |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Coding Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DVC] | Using consistent values in a program improves readability and is less prone to errors, which can not only compromise the integrity of the program itself but also create security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Noncompliant code would use unexplained values that are hard-coded, making the code less understandable. |
| #include <iostream>  // Noncompliant code using magic number  int main() {  int radius = 8; // Using hard-coded numbers without explanation  double areaOfCircle = 3.14 \* radius \* radius;  std::cout << "Circle area: " << areaOfCircle << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Compliant code would use constants that are understandable. |
| #include <iostream>  const double PI = 3.14;  const int RADIUS = 8;  // Compliant code using symbolic constants  int main() {  double areaofCircle = PI \* RADIUS \* RADIUS;  std::cout << "Circle area: " << areaOfCircle << std::endl;  return 0;  } |

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

| **Principles(s):**  (1: Validate Input Data: Ensuring that data values are consistent and conform to a specific criteria can be done by validating input data. Validating data type by adhering to predetermined ranges or types prevents the integrity of the program from being compromised, thereby reducing vulnerabilities.)  (4: Keep It Simple: Reducing complexity of the program by using consistent values ensures the program’s maintainability. It also reduces errors, creating a more secure program overall.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.01 | RESOURCE\_LEAK | Static analysis tool to detect code issues that includes resource leaks, memory, and resource management. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness** |
| --- | --- | --- |
| **String Correctness** | [STD-003-SCC] | String correctness ensures strings to be handed and manipulated properly, preventing errors or vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Noncompliance would entail unintentional buffer overflows because of insecure string operations. |
| #include <iostream>  #include <cstring>  // Noncompliant code with insecure string operation  int main() {  char str[10];  strcpy(str, "Hello, World!"); // Not checking bounds, creating buffer overflow  std::cout << str << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Compliant code checks the bounds and uses secure string functions. It prevents buffer overflows by ensuring the destination buffer is null-terminated. |
| #include <iostream>  #include <cstring>  const int MAX\_LENGTH = 20;  // Compliant code with secure string operation  int main() {  char str[MAX\_LENGTH];  strncpy(str, "Hello, World!", MAX\_LENGTH - 1); // bounds checking with strncpy  str[MAX\_LENGTH - 1] = '\0'; // Null-terminate the string  std::cout << str << std::endl;  return 0;  } |

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

| **Principles(s):**  (1: Validate Input Data: Validating strings promotes the proper handling and manipulation of string data. Adhering to predetermined limits (format, length, etc.) prevents errors, thus preventing vulnerabilities.)  (7: Sanitize Data Sent to Other Systems: Ensuring string data is sanitized before transferring it to other systems reduces the risk of vulnerabilities to injection attacks and unintended behavior of the code, which creates a more secure program.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | -Wstring  -conversion | Static analysis tool that detects issues in C/C++ code, including string conversion problems, preventing runtime errors. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
|  |  |  |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | SQL Injection prevention creates a barrier against injection attacks by sanitizing input. |

| **Noncompliant Code** |
| --- |
| Noncompliant code embeds user input into SQL queries directly, which makes the code vulnerable to injection attacks. This code concatenates user input, a username and password, into an SQL query. |
| #include <iostream>  #include <string>  int main() {  std::string username = "admin";  std::string password = "password";    // Concatenates user input into SQL query  std::string query1 = "SELECT \* FROM users WHERE username='" + username + "' AND password='" + password + "'";    //execute    return 0;  } |

| **Compliant Code** |
| --- |
| This code parameterizes queries or uses prepared statements that safely handles input. |
| #include <iostream>  #include <string>  #include <asql/asql.h> // Example SQL library  // Compliant code using parameterized query  int main() {  std::string username = "admin";  std::string password = "password";    // Initialize SQL connection and statement  MYSQL \*connection;  MYSQL\_STMT \*stmt = asql\_stmt\_init(connection);    // parameterize query  std::string query = "SELECT \* FROM users WHERE username=? AND password=?";  asql\_stmt\_prepare(stmt, query.c\_str(), query.length());    // Bind parameters  SQL\_BIND bindParams[2];  bindParams[0].buffer\_type = MYSQL\_TYPE\_STRING;  bindParams[0].buffer = (void \*)username.c\_str();  bindParams[0].buffer\_length = username.length();  bindParams[0].is\_null = 0;  bindParams[1].buffer\_type = MYSQL\_TYPE\_STRING;  bindParams[1].buffer = (void \*)password.c\_str();  bindParams[1].buffer\_length = password.length();  bindParams[1].is\_null = 0;  asql\_stmt\_bind\_param(stmt, bindParams);    // Execute    return 0;  } |

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

| **Principles(s):**  (1: Validate Input Data: SQL injection attacks can be prevented by validating input data. Sanitizing input data rejects malicious SQL code from being injected into the program by ensuring the input data conforms to predefined criteria.)  (7: Sanitize Data Sent to Other Systems: Sanitizing data before sending it across systems prevents SQL injection attacks by rejecting input that the program could read as an SQL command, reducing vulnerabilities.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.2 | sql-injection | Static analysis tool to detect security issues, including SQL injection vulnerabilities in C/C++ code. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
|  |  |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MPC] | Protecting memory prevents vulnerabilities such as dangling pointers, leaks, and overflows. |

| **Noncompliant Code** |
| --- |
| This code demonstrates noncompliance improper management by having a dangling pointer, created by deleting the pointer. |
| #include <iostream>  // dangling pointer  int main() {  int \*ptr = new int(9);  delete ptr;  // Accessing a dangling pointer  std::cout << \*ptr << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code uses smart pointers to deallocate memory when necessary. |
| #include <iostream>  #include <memory>  // smart pointers for memory management  int main() {  std::unique\_ptr<int> ptr = std::make\_unique<int>(9);  // Accessing smart pointer  std::cout << \*ptr << std::endl;  return 0;  } |

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

| **Principles(s):**  (2: Heed Compiler Warnings: Compiler warnings flag for memory protection vulnerabilities such as buffer overflows, leaks, or dangling pointers. Heeding these warnings will reduce risk related to memory and ensure security.)  (4: Keep It Simple: Reducing complexity in turn reduces the need for complex memory management. Complex memory management can lead to vulnerabilities and is more difficult to read and maintain. Keeping memory management simple will keep the program more stable and secure.)  (8: Practice Defense in Depth: Defending against vulnerabilities using a multi-layered system of security enhances memory protection and in turn enhances overall security of the program.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | memcheck | Dynamic analysis tool to detect memory-related issues like leaks, improper management. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASC] | Assertions detect unexpected behavior by validating assumptions and conditions to prevent runtime errors. |

| **Noncompliant Code** |
| --- |
| This noncompliant code does not use any assertions to check if the function to divide numbers is dividing by zero, which could lead to errors. |
| #include <iostream>  // without assertions  int divide(int x, int y) {  // not checked  return x / y;  }  int main() {  int result = divide(10, 0); // divide by zero  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code includes an assertion to check for a division by 0, throwing an error to prevent unintended behavior. |
| #include <iostream>  #include <cassert>  // with assertions  int divide(int x, int y) {  assert(y != 0); // Assert division by zero  return x / y;  }  int main() {  int result = divide(10, 0); // failure: Division by zero  std::cout << "Result: " << result << std::endl;  return 0;  } |

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

| **Principles(s):**  (1: Validate Input Data: Validating input data with assertions can add another layer of security by catching input that does not adhere to predefined criteria that would otherwise cause runtime errors or other unexpected behavior, enhancing stability and security of the program.  (9: Use Effective Quality Assurance Techniques: Assertions placed throughout the program can routinely verify that the program is adhering to predefined conditions and assumptions, catching unexpected behavior. This ensures a stable and reliable program by reducing exploitable vulnerabilities and enhancing the program quality.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.3 | enable=warning  style  performance  portability | Static analysis tool to detect potential issues including incorrect or unused assertions. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | Handling exceptions manages runtime errors and prevents unintended program behavior. |

| **Noncompliant Code** |
| --- |
| Noncompliant code may have exceptions but without any proper handling, which would still result in unintended behavior. |
| #include <iostream>  // without exception handling  int divide(int x, int y) {  if (y == 0) {  // Division by zero  throw "Division by zero";  }  return x / y;  }  int main() {  try {  int result = divide(10, 0); // Potential error  std::cout << "Result: " << result << std::endl;  } catch (const char\* message) {  std::cerr << "Error: " << message << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code throws a specific exception type, is caught and throws a specific message, reducing runtime errors and unexpected behavior. |
| #include <iostream>  #include <stdexcept>  // exception handling  int divide(int x, int y) {  if (y == 0) {  // error with an exception  throw std::invalid\_argument("Division by zero");  }  return x / y;  }  int main() {  try {  int result = divide(10, 0); // division by zero  std::cout << "Result: " << result << std::endl;  } catch (const std::invalid\_argument& ex) {  std::cerr << "Error: " << ex.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):**  (4:Keep It Simple: Reducing complexity prevents unexpected behaviors often caused by unhandled exceptions. Properly handling them with in-depth exception handling practices can ensure readability and prevent errors that could let vulnerabilities slip through the cracks.)  (8: Practice Defense in Depth: A robust defense in depth policy should include exception handling, which will manage runtime errors, unexpected behaviors, and enhance the prevention of exploitable vulnerabilities.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.01 | EXCEPTIONS | Static analysis tool to detect potential issues like improper exception handling or uncaught exceptions. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Modifying String Literals | [STD-008-MSL] | Modifying string literals directly can lead to unintended behavior. |

| **Noncompliant Code** |
| --- |
| The following code modifies the string literal directly. |
| #include <iostream>  int main() {  char\* str = "Hello, World!";  str[0] = 'x'; // Modifying a string literal directly  std::cout << str << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code uses a modifiable array, instead of a direct string literal. |
| #include <iostream>  #include <cstring>  int main() {  char str[] = "Hello, World!"; // modifiable array  str[0] = 'x'; // Modify the array content  std::cout << str << std::endl;  return 0;  } |

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

| **Principles(s):**  (3: Architect and Design for Security Policies: Implementing secure coding practices such as the prohibition or limitation of modifying string literals prevents the introduction of vulnerabilities. This maintains stability and the integrity of the program.)  (6: Adhere to the Principle of Least Privilege: Adhering to the principle of least privilege limits access to modifying string literals which prevents unintended behavior caused by direct modification. Limiting access to include only the users or functions necessary to run the program ensures a more secure program.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 20.2.0 | STRING\_MODIFICATION | Static analysis tool to identify security vulnerabilities in C/C++ code, like issues that arise from modifying string literals. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Do Not Access Freed Memory | [STD-009-FMC] | Accessing memory that is deallocated can lead to unintended behavior which can include data corruption, vulnerabilities, and crashes. |

| **Noncompliant Code** |
| --- |
| The code snippet accesses freed memory directly using a pointer, which can cause the program to crash. |
| #include <iostream>  int\* allocateMemory() {  return new int(34);  }  void useMemory() {  int\* ptr = allocateMemory();  delete ptr;  std::cout << \*ptr << std::endl; // Accessing freed memory  }  int main() {  useMemory();  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code uses allocated memory, then deallocating memory directly after its access, preventing attempts to access the freed memory. |
| #include <iostream>  int\* allocateMemory() {  return new int(34);  }  void useMemory() {  int\* ptr = allocateMemory();  std::cout << \*ptr << std::endl; // Using allocated memory  delete ptr; // Deallocating memory directly after  }  int main() {  useMemory();  return 0;  } |

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

| **Principles(s):**  (2: Heed Compiler Warnings: Compiler warnings are useful in the detection of errors related to the accessing of freed memory. Heeding these warnings prevents runtime errors and other unintended behavior that accessing deallocated memory may cause.  (8: Practice Defense in Depth: Defense in depth should always include the denial of access to freed memory. Using memory-safe languages, nullifying pointers and other memory management strategies can prevent vulnerabilities, data corruption, unintended behavior.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Dynamic analysis tool to detect issues such as accessing freed memory. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Call Functions With Correct Arguments | [STD-010-FAC] | Call functions with correct arguments to ensure stability. Failing to do so can result in corruption of memory and stack, or other unintended behavior. |

| **Noncompliant Code** |
| --- |
| The noncompliant code demonstrates calling a function with the incorrect number of arguments. Doing the same with incorrect type of argument will also result in error. |
| #include <iostream>  void printSum(int n, int x) {  std::cout << "Sum: " << n + x << std::endl;  }  int main() {  int s = 10;  printSum(s); // Incorrect number of arguments  return 0;  } |

| **Compliant Code** |
| --- |
| Compliant code will call the correct number and type of arguments, which will prevent errors and improve reliability. |
| #include <iostream>  void printSum(int n, int x) {  std::cout << "Sum: " << n + x << std::endl;  }  int main() {  int s = 10, y = 20;  printSum(s, y); // Correct number of arguments  return 0;  } |

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

| **Principles(s):**  (1: Validate Input Data: Accuracy in function calls is directly related to validating input. Validating input data before it is passed through to the function calls prevents unintended behavior such as memory corruption or leaks, ensuring a more secure program.)  (8: Practice Defense in Depth: A solid defense in depth strategy involves ensuring functions are called correctly, which prevents memory corruption which in turn can cause unintended behavior. This creates a more stable program, preventing vulnerabilities.) |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | -Wextra  -argument | Static analysis tool to detect issues in C/C++ code, like incorrect function arguments. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

Automating the enforcement of the standards in the security policy should be integrated throughout the entire process and infrastructure. Leaving security to a separate stage or until the end of the software development life cycle not only increases the risk of leaving vulnerabilities, but could also potentially increase the time until the launch of a program.

In the access and planning phase, project requirements should include security requirements. This includes secure coding practices such as data type validation and input sanitization. User stories and acceptance criteria should also consider security. During the design and build phases, security checks should be integrated. This is where static code analysis tools be implemented into the IDEs to provide immediate feedback regarding security issues. The verify and test phase should have automated security testing integrated into it. Static and dynamic analysis tools for runtime testing can identify potential vulnerabilities. Test cases that are security focused should be included. This is also where the code should be submitted for a security review before it is released to production.

In the transition and health check phase, controls should be integrated to securely configure the environment. Access controls and encryption should be implemented in this phase. Continuous monitoring and logging should be implemented in the monitor and detect phase. This can catch attacks in real time. Security information and event management (SIEM) tools can automate the detection of suspicious activity. The response phase can have procedures implemented to ensure a timely and organized response to threats and events.

### 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-DTC] | High | Medium | Medium | High | 4 |
| [STD-002-DVC] | High | Medium | Medium | High | 4 |
| [STD-003-SCC] | High | Medium | Medium | High | 4 |
| [STD-004-SQL] | High | High | High | High | 5 |
| [STD-005-MPC] | High | High | High | High | 5 |
| [STD-006-ASC] | Medium | Medium | Low | Medium | 3 |
| [STD-007-EXC] | High | High | High | High | 5 |
| [STD-008-MSL] | High | Medium | Medium | High | 4 |
| [STD-009-FMC] | High | High | High | High | 5 |
| [STD-010-FAC] | High | Medium | Medium | High | 4 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 | Used to encrypt data during storage on devices such as databases, cloud servers, or hard drives. This prevents unauthorized access by using algorithms to encrypt entire disks. This applies whenever sensitive data is stored. It ensures that the data remains protected if the storage device is accessed without authorization. |
| Encryption in flight | Used to secure data during transmission over a network, ensuring confidentiality and prevents interception. It is used to secure systems and communication between them such as servers or applications and databases. This is applied whenever data is transmitted over untrusted networks. It ensures the confidentiality of sensitive data. |
| Encryption in use | Used to encrypt data while it is being used by an application. The data stays encrypted using cryptographic libraries or APIs. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users accessing a system or application. Authentication usually uses passwords, biometrics, multi-factor authentication (MFA), and single sign-on (SSO). It also controls access to systems or data based on the identity of users. This applies whenever access control is required to protect the integrity sensitive data. |
| Authorization | Authorization determines the permissions given to verified users. This employs the principle of least privilege, using control lists (ACLs), attribute-based access control (ABAC), and role-based access control (RBAC). |
| Accounting | Accounting traces and records user activity involving a system. It ensures security by using audit trails, logging, etc. This helps detect malicious use of a system and applies whenever a user is accessing a system. |

**\***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 |  |
| [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 |