**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 | All inputs need to be checked using a proven method of validation. This is to prevent any security vulnerabilities, including buffer overflow, SQL injection, XSS, and other attacks. All forms of input are considered unsafe until properly validated. |
| 1. Heed Compiler Warnings | During the build process, always heed compiler warnings. They provide great insight into potential security risks. This may include data type mismatches, uninitialized variables, and runtime buffer overflow detection. |
| 1. Architect and Design for Security Policies | Projects must be built using a robust set of security policies. These policies are to be incorporated throughout the entire development lifecycle. These policies may include principle of least privilege, input validation, data sanitization, etc. |
| 1. Keep It Simple | Code must keep to best modern practices, building quality, clear, and efficient code. This includes keeping variable names clean and descriptive, functions follow the single responsibility principle, and documentation is clear and concise. |
| 1. Default Deny | All permissions are to be denied unless otherwise deemed acceptable by the authorization policy. This approach is generally used in firewalls to deny all inbound and outbound traffic unless permissible by the firewall policy. |
| 1. Adhere to the Principle of Least Privilege | Each user should have just enough privilege to accomplish their given tasks. This policy is used to prevent users from accessing impermissible areas with malicious intent. |
| 1. Sanitize Data Sent to Other Systems | Any data that is passed to another system must be validated. An example of this is sanitizing SQL queries before executing them in the database. All queries must be checked for SQL injection attempts before making queries. |
| 1. Practice Defense in Depth | Security must be layered and spread over the entire system. These layers may include physical, network, application, and procedures. The purpose behind defense in depth is to leverage multiple security measures to protect the company’s assets. |
| 1. Use Effective Quality Assurance Techniques | Utilize robust testing libraries, static and dynamic analysis tools for checking dependencies, exceptions to handle errors gracefully, and assertions to test for expressions. Incorporating these techniques will generate quality code. |
| 1. Adopt a Secure Coding Standard | It’s important to adopt a secure coding standard like SEI CERT C++ Coding Standard, which illustrates several examples of using components and techniques safely like input/output, concurrency, containers, integers, expressions, and much more. |

### 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] | Declare variables with the appropriate data type. Choose a data type that fits the given value range. Additionally, unsigned values should be used for values that do not go into the negative. |

| **Noncompliant Code** |
| --- |
| The iterator, *i,* is being declared using unsigned long int, which is way too big for the given value range. |
| 1 const unsigned char MAX\_SIZE = 10;  2 unsigned long int sum = 0;  3 unsigned char arr[MAX\_SIZE] = /\* initialize \*/;  4  5 for (unsigned long int i = 0; i < MAX\_SIZE; i++) {  6 sum += arr[i];  7 } |

| **Compliant Code** |
| --- |
| The iterator, *i,* is being declared using an unsigned char because it will always be positive and stay within the given range. |
| 1 const unsigned char MAX\_SIZE = 10;  2 unsigned long int sum = 0;  3 unsigned char arr[MAX\_SIZE] = /\* initialize \*/;  4  5 for (unsigned char i = 0; i < MAX\_SIZE; i++) {  6 sum += arr[i];  7 } |

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

| **Principles(s):**  #2 Data types must be appropriate for value range  #4 Explicit variables are easier to debug than implicit declarations |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | UNLIKELY | LOW | MEDIUM | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Eclair | 3.12.0 | [DCL31-C](https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them) | Checks for potential overflow/underflow and implicit declarations |
| Astree | 22.10 | Data type specifiers  Implicit declarations  Uninitialized declarations | Checks for potential overflow/underflow |
| Polyspace Bug Finder | R2022b | [DCL31-C](https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them) | Detects implicit declarations |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Data values should fall within the given range of the declared data types to prevent wrapping. When large data types are casted to smaller data types, this may result in truncation. |

| **Noncompliant Code** |
| --- |
| This casting may result in largeValue being truncated because smallValue is declared using a much smaller data type. |
| 1 unsigned long int largeValue = ULONG\_MAX;  2 signed int smallValue;  3 smallValue = (signed int) largeValue; |

| **Compliant Code** |
| --- |
| Check if value is in range before casting to new data type. If not, handle the error. |
| 1 unsigned long int largeValue = ULONG\_MAX;  2 unsigned int smallValue;  3  4 if (largeValue <= SINT\_MAX) {  5 smallValue = (signed int) largeValue;  6 } else {  7 /\* handle error \*/  8 } |

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

| **Principles(s):**  #1 Values must be kept within range for given data type  #8 Initialized variables strengthen DiD |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | MEDIUM | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | [EXP53-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory) | Warnings for uninitialized variables |
| Polyspace Bug Finder | R2022b | [EXP53-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory) | Checks for buffer overflow/underflow |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee the string size does not result in a buffer overflow. Additionally, make sure the string contains enough space for the null terminator. |

| **Noncompliant Code** |
| --- |
| The input is unbounded and can result in a buffer overflow. |
| 1 #include <iostream>  2  3 int main(void) {  4 char buf[10];  5 std::cin >> buf;  7 } |

| **Compliant Code** |
| --- |
| By setting the width field of std::cin to 10, it prevents a buffer overflow from occurring. |
| 1 #include <iostream>  2  3 int main(void) {  4 char buf[10];  5  6 std::cin.width(10);  7 std::cin >> buf;  8 } |

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

| **Principles(s):**  #1 Use data type with enough space for null terminator  #7 Preventing buffer overflows helps eliminate SQL injection  #8 Modern string operations utilize null terminator and bound checking which strengthens DiD |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1 | [STR50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) | Checks for missing null terminator and buffer overflow |
| Polyspace Bug Finder | R2022b | [STR50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) | Checks for null terminator and buffer overflow |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | To prevent SQL injection, it’s best to use prepared statements for querying. This prevents any data being passed in to execute unwanted commands in SQL. |

| **Noncompliant Code** |
| --- |
| This code does not use prepared statements. Instead, it directly injects the user input into the SQL query string. |
| 1 uName = getRequestString("username");  2 uPass = getRequestString("userpassword");  3  4 sql = “SELECT \* FROM Users WHERE Name = " + uName + " AND Pass = " +  5 uPass + ” |

| **Compliant Code** |
| --- |
| Using prepared statements ensures the data being passed in acts as a string, not a command. |
| 1 PreparedStatement pStmt = PreparedStatement();  2  3 std::cin >> username;  4 std::cin >> userpassword;  5  6 sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username,  7 userpassword}; |

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

| **Principles(s):**  #3 Utilize prepared statements and parameterized queries  #7 Sanitize input before performing SQL queries |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | [FIO30-C](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings) | Provides taint analysis |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Accessing memory after it has been freed can cause a program to crash, responding with an access violation. |

| **Noncompliant Code** |
| --- |
| Attempting to access memory after it has been freed will result in an access violation. |
| 1 #include <stdio.h>  2 #include <stdlib.h>  3 #include <string.h>  4  5 int main (int argc, char \*\* argv) {  6 char \* p = malloc(10);  7 free(p);  8 \*p = 10; /\* results in an access violation \*/  9 return 0;  10 } |

| **Compliant Code** |
| --- |
| Ensuring that all mutations on memory must be performed before it has been freed will help protect memory. |
| 1 #include <stdio.h>  2 #include <stdlib.h>  3 #include <string.h>  4  5 int main (int argc, char \*\* argv) {  6 char \* p = malloc(10);  7 \*p = 10; /\* does not result in an access violation \*/  8 free(p);  9 return 0;  10 } |

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

| **Principles(s):**  #3 Free memory that is no longer needed  #9 Utilize static analysis tools and QA techniques to identify potential memory bugs |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | [MEM50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory) | Checks for access of freed memory |
| Polyspace Bug Finder | R2022b | [MEM50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory) | Checks for potential out-of-bounds index and access of freed memory |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions will test expressions to ensure proper conditions are met. If the test fails, the abort() function is called, preventing erroneous behavior. |

| **Noncompliant Code** |
| --- |
| There is no validation for myInt in display\_number() to ensure the value is not NULL before printing the value. |
| 1 #include <iostream>  2 #include <cassert>  3 using namespace std;  4  5 void display\_number(int\* myInt) {  6   // prints value  7   cout << "myInt contains value" << " = " << \*myInt << endl;  8 }  9  10 int main ()  11 {  12   int myptr = 5;  13   int \* second\_ptr = NULL;  14   int \* third\_ptr = NULL;  15   second\_ptr = &myptr;  16   display\_number (second\_ptr);  17   display\_number (third\_ptr);  18   return 0;  19 } |

| **Compliant Code** |
| --- |
| Using assert prevents printing from unreferenced memory. |
| 1 #include <iostream>  2 #include <cassert>  3 using namespace std;  4  5 void display\_number(int\* myInt) {  6   assert (myInt != NULL);  7   cout << "myInt contains value" << " = " << \*myInt << endl;  8 }  9  10 int main ()  11 {  12   int myptr = 5;  13   int \* second\_ptr = NULL;  14   int \* third\_ptr = NULL;  15   second\_ptr = &myptr;  16   display\_number (second\_ptr);  17   display\_number (third\_ptr);  18   return 0;  19 } |

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

| **Principles(s):**  #9 Implement assertions and other tests to check software conditions |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | MEDIUM | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft Tests | 2022.1 | [MSC11-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions) | Proper usage of assertions, provides warnings for assignments within assertions and potential side effects |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Any time the called function is unable to complete its work, an exception should be thrown to handle the error gracefully. An example of using exceptions is to check for errors in input validation or allocating memory. |

| **Noncompliant Code** |
| --- |
| The instantiation of the myArr variable may cause the program to crash if there is not enough memory. |
| 1 #include <iostream>  2 #include <exception>  3 using namespace std;  4  5 int main ()  6 {  7    int\* myArr = new int[1000];  8 return 0;  9 } |

| **Compliant Code** |
| --- |
| The given try block checks the allocation of memory. If it fails, a bad\_alloc exception is thrown and handled in the catch block. |
| 1 #include <iostream>  2 #include <exception>  3 using namespace std;  4  5 int main ()  6 {  7   try {  8    int\* myArr = new int[1000];  9 }  10 catch (exception& e) {  11 cout << “Standard exception: “ << e.what() << endl;  12 }  13 return 0;  14 } |

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

| **Principles(s):**  #3 Capture exception-throwing code in try-catch-finally blocks  #9 Check for potential exceptions to ensure the software exits gracefully |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | UNLIKELY | MEDIUM | HIGH | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | [ERR56-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR56-CPP.+Guarantee+exception+safety) | Catch all possible exceptions |
| Polyspace Bug Finder | R2022b | [ERR56-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR56-CPP.+Guarantee+exception+safety) | Catch all unhandled exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-008-CPP] | Ensure that operations performed on integer values do not result in an overflow. One way to prevent this is to check the bounds of the value before performing an operation. |

| **Noncompliant Code** |
| --- |
| The following code increments the result value without checking if it reaches its upper bound, possibly resulting in an integer overflow. |
| 1 template <typename T>  2 T add\_numbers(T const& start, T const& increment, unsigned long int const& 3 3 steps)  4 {  5 T result = start;  6 for (unsigned long int i = 0; i < steps; ++i)  7 {  8 result += increment;  9 }  10  11 return result;  12 } |

| **Compliant Code** |
| --- |
| The following code checks the bounds of the data type before incrementing the value, preventing the data from overflowing. It checks if the amount to be added is greater than the difference between the max data type value and the current value. |
| 1 template <typename T>  2 T add\_numbers(T const& start, T const& increment, unsigned long int const& 3 steps)  4 {  5 T result = start;  6 for (unsigned long int i = 0; i < steps; ++i)  7 {  8 // check if data type will overflow when adding final increment  9 // if overflow is detected, return a zero  10 if ((increment > 0 && result > std::numeric\_limits<T>::max() - increment) ||  12 (increment < 0 && result < std::numeric\_limits<T>::min() – increment)) {  14 return 0;  15 }  16 // otherwise, increment result  17 else {  18 result += increment;  19 }  20 }  21  22 return result;  23 } |

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

| **Principles(s):**  #1 / #9 Verify integer operations to prevent overflow/underflow  #7 Verify dynamic input before using in operations |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | MEDIUM | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | [INT02-C](https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules) | Integer overflows, conversion checks, and integer operations |
| Polyspace Bug Finder | R2022b | [INT02-C](https://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules) | Checks integer overflow/underflow, wrapped integers, conversions |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input / Output | [STD-009-CPP] | Make sure to close files that are associated with stream objects after they are no longer needed. In larger programs, failing to do so can use up resources and block other processes from acquiring that file. Additionally, check if the file failed to open and failed to close. |

| **Noncompliant Code** |
| --- |
| The following code opens a file, checks if the file failed to open, handles the error if so, but does not attempt to close the file associated with the stream object. |
| 1 #include <exception>  2 #include <fstream>  3 #include <string>  4  5 void f(const std::string &fileName) {  6 std::fstream file(fileName);  7 if (!file.is\_open()) {  8 /\* handle error \*/  9 return;  10 }  11 } |

| **Compliant Code** |
| --- |
| This solution attempts to close the file after use, check if the closing failed, and handles the error if it does. |
| 1 #include <exception>  2 #include <fstream>  3 #include <string>  4  5 void f(const std::string &fileName) {  6 std::fstream file(fileName);  7 if (!file.is\_open()) {  8 /\* handle error \*/  9 return;  10 }  11  12 file.close();  13 if (file.fail()) {  14 /\* handle error \*/  15 }  16 } |

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

| **Principles(s):**  #7 Validate dynamic input values before performing operations on them  #9 / #10 Close files when no longer needed |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | UNLIKELY | LOW | HIGH | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1 | [FIO51-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed) | Checks for data leaks |
| Polyspace Bug Finder | R2022b | [FIO51-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed) | Checks for data leaks |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-010-CPP] | Do not try to read uninitialized memory. This may result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| Attempting to print uninitialized values can result in undefined behavior. |
| 1 #include <iostream>  2  3 void f() {  4   int i;  5   std::cout << i;  6 } |

| **Compliant Code** |
| --- |
| The variable is initialized to a value before attempting to print. |
| 1 #include <iostream>  2  3 void f() {  4   int i = 0;  5   std::cout << i;  6 } |

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

| **Principles(s):**  #2 Initialize variables  #7 Validate dynamic input before performing operations on them |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | LIKELY | LOW | MEDIUM | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Atree | 22.10 | [EXP33-C](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory) | Checks non initialized variables |
| Polyspace Bug Finder | R2022b | [EXP33-C](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory) | Checks non initialized variables and pointers |

### 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.

Each phase in the DevOps chain will contain additional measures to enforce the new standards in this policy. The pre-production phases will be adding security tool training and in-depth threat analysis to the assess and plan phase. Both design and build phases will incorporate additional security measures within the IDE. The verify and test phase will consist of static and dynamic testing techniques, like unit, system, and integration.

Once the project has entered the production phase, testing will be emphasized to monitor and detect issues within the program. Performance metrics are captured and logged for developers to adjust and eliminate bottlenecks. Penetration testing is performed to identify any new potential security threats. Additional feedback concerning bugs and performance issues is also welcomed from end users.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | LOW | Unlikely | LOW | MEDIUM | 2 |
| STD-002-CPP | HIGH | LIKELY | MEDIUM | MEDIUM | 4 |
| STD-003-CCP | HIGH | LIKELY | MEDIUM | HIGH | 4 |
| STD-004-CCP | HIGH | LIKELY | MEDIUM | HIGH | 5 |
| STD-005-CCP | HIGH | LIKELY | MEDIUM | HIGH | 5 |
| STD-006-CCP | HIGH | LIKELY | HIGH | MEDIUM | 2 |
| STD-007-CCP | LOW | UNLIKELY | MEDIUM | HIGH | 4 |
| STD-008-CCP | HIGH | LIKELY | HIGH | MEDIUM | 4 |
| STD-009-CCP | MEDIUM | UNLIKELY | LOW | HIGH | 3 |
| STD-010-CCP | MEDIUM | LIKELY | LOW | MEDIUM | 2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Ensures to protect data that is stored. This may include storage media like mechanical drives, solid-state drives, databases, or cloud. Data at rest must be kept protected from security exploits, data breaches, and physical threats. |
| Encryption at flight | Ensures to protect data is being transmitted from one place to another. This is typically over a network via the web. It’s important to utilize security protocols like SSL/TLS/HTTPS when transmitting data across a network. This is to prevent threats like eavesdropping and man-in-the-middle attacks. |
| Encryption in use | Ensures to protect data that is being accessed, processed, viewed, etc. Data in use occurs between encryption at rest and encryption in flight. Memory protection techniques are required to prevent unwanted access to memory, potentially exposing sensitive information like encryption keys. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of a user. Popular methods of authentication include username-password combinations, biometrics, and two-step verification. Without this step, malicious users are able to steal the personal identifiable information of others. |
| Authorization | Once the user is authenticated, the permissions of that user must be determined. That is the purpose of authorization. The principle of least privilege is commonly used to make sure users have the minimum amount of privilege required to accomplish their given tasks. |
| Accounting | Accounting is used to log information in records and file to keep track of common transactions. These typically include events like user creation, account updates, transfers, and logins. This allows administrators to see what is happening at all times to ensure authentication and authorization policies are working properly. |

**\***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 | 11/09/2022 | Initial Template (M3) | Joshua Eberharter |  |
| 2.0 | 11/23/2022 | Module Five Milestone | Joshua Eberharter |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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