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

**Quintin B. Rozelle**



# 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 | Input that originates outside of the program cannot be trusted. All external data must be validated prior to acting upon to ensure it will not present a security threat (e.g., SQL injection, buffer overflow, etc.). |
| 1. Heed Compiler Warnings | The complier presents warnings when it discovers issues with the code. There are times when these issues are not severe enough to cause compilation errors, but do introduce a potential security gap (e.g., deprecated functions, etc.). Reviewing compiler warnings and working to eliminate these can help to make code more secure. |
| 1. Architect and Design for Security Policies | Architecture design and security policies exist to provide guidance on optimal coding strategies. Failure to follow these guidelines can result in security flaws in code which must be avoided. |
| 1. Keep It Simple | Simple code is easier to understand, maintain, and fix. This helps to find and correct security flaws when they present themselves. As described in the “Zen of Python”, simple is better than complex, complex is better than complicated. |
| 1. Default Deny | Denial of access should be the default action. If it is not possible to confirm a user’s credentials and access rights, it is better to prevent access and potential security breaches than to assume good intent and introduce a potential bad actor to the environment. |
| 1. Adhere to the Principle of Least Privilege | Always provide the minimal amount of access rights necessary to users. Any more than needed to perform their duties opens the door to accidental or intentional security breaches. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems ensures that unnecessary information is removed an irretrievable from outside sources. This protects the sensitive information from theft and limits a bad actor’s ability to review that data for clues on gaining unauthorized access to secure systems. |
| 1. Practice Defense in Depth | No one defensive method is capable of preventing all attacks and security breaches. Incorporating multiple lines of defense increases the chances that at least one layer is capable of blocking and preventing a security threat. |
| 1. Use Effective Quality Assurance Techniques | Thorough quality assurance practices will help to ensure that code is written securely and without errors. This can encompass static and dynamic testing, vulnerability testing, edge case tests, and employing white hat hackers (among many other techniques). Incorporating these methods can help to limit insecure coding errors. |
| 1. Adopt a Secure Coding Standard | A coding standard can help to ensure that all individuals involved with code development are on the same page and are incorporating similar coding practices. This helps to make code more manageable and helps to prevent security vulnerabilities. |

### 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] | Ensure that conversions between integer types don’t result in lost data.  Because different integer types are capable of holding different maximum values, converting from a larger type to a smaller can result in lost data. |

| **Noncompliant Code** |
| --- |
| This code example will result in lost data upon type conversion if liA is greater than the maximum possible value for a short int. |
| short convertLongToShort(long liA) {  short siB = (short)liA;  return siB;  } |

| **Compliant Code** |
| --- |
| The following compliant code example provides a check prior to type conversion to ensure the value supplied to the function isn’t too large to convert without data loss. |
| #include <limits.h>  short convertLongToShort(long liA) {  if (liA <= SHRT\_MAX) {  short siB = (short)liA;  return siB;  } else {  // Data loss would happen. Handle error  }  } |

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

| **Principles(s):**  1 - Often times, user input as a number must be converted to different integer types. Validating that the user input can be correctly converted between types without loss of information is necessary.  9 – Ensuring that integer conversions don’t cause loss of data can help to make a program that functions as intended.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | [memsetValueOutOfRange](https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/) | Checks that the second argument for memset() isn’t an unsigned char. |
| Compass/ROSE |  |  | Inclusion of limits.h can cause false warnings. |
| Polyspace Bug Finder | R2024a | [CERT C: Rule INT31-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint31c.html) | Checks for integer overflow. |
| RuleChecker | 20.04 |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensure that operations on unsigned integers do not wrap.  An operation that exceeds the bounds of an unsigned integer will present unexpected results as the value of that integer will wrap (e.g., exceeding the upper bound will result in the integer taking its minimum possible value). |

| **Noncompliant Code** |
| --- |
| This code example can result in wrapping if the addition of uiA and uiB results in a value that is greater than an integers maximum possible value. In a case like this, the function would return an unexpectedly low value. |
| int addIntegers(unsigned int uiA, unsigned int uiB) {  return uiA + uiB;  } |

| **Compliant Code** |
| --- |
| The following compliant code example provides a check prior to addition to ensure that wrapping doesn’t occur. If wrapping is detected, it is handled appropriately in the first part of the if statement. |
| #include <limits.h>  int addIntegers(unsigned int uiA, unsigned int uiB) {  if (UINT\_MAX – uiA < uiB) {  // Wrapping would occur. Handle appropriately  } else {  return uiA + uiB;  }  } |

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

| **Principles(s):**  1 – User input must be validated that operations performed on it will not cause integer wrapping.  3 – Integer wrapping can cause unintended behavior and security risks. Design the code to prevent these issues.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | [CERT C: Rule INT30-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint30c.html) | Checks for integer overflow. |
| PVS-Studio | 7.32 | [V658](https://pvs-studio.com/en/docs/warnings/v658/), [V1012](https://pvs-studio.com/en/docs/warnings/v1012/), [V1028](https://pvs-studio.com/en/docs/warnings/v1028/), [V5005](https://pvs-studio.com/en/docs/warnings/v5005/), [V5011](https://pvs-studio.com/en/docs/warnings/v5011/) |  |
| Coverity | 2017.07 | INTEGER\_OVERFLOW |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensure that the space allotted for a string is enough to contain the characters and terminator.  C-style strings require a null terminator to notate the end of a string. If the space allotted for the string isn’t large enough to include the string data and a null terminator, security errors can happen as the code is unable to correctly determine the end of the string. Note: std::basic\_string manages it’s own memory and is preferable to using C-style strings. |

| **Noncompliant Code** |
| --- |
| This code example can result in a buffer overflow if more than 19 characters are read from std::cin. |
| #include <iostream>  void getUserInput() {  char str[20];  std::cin >> str;  } |

| **Compliant Code** |
| --- |
| The following compliant code example uses the std::basic\_string class which manages it’s own memory. When used correctly, this class will help to prevent issues present with C-style strings. |
| #include <iostream>  #include <string>  void getUserInput() {  std::string str;  std::cin >> str;  } |

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

| **Principles(s):**  1 – Validate string input to ensure that it will completely fit within the space allotted for it.  3 – Design code to prevent buffer overflows which can lead to security risks.  10 – A secure coding standard can prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Checks for buffer overflow from unsafe string functions. |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE |  |
| Polyspace Bug Finder | R2024a | [CERT C: Rule STR31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr31c.html) | Checks for missing null terminator, buffer overflows, and inadequate destination buffer sizes. |
| PVS-Studio | 7.32 | [V518](https://pvs-studio.com/en/docs/warnings/v518/), [V645](https://pvs-studio.com/en/docs/warnings/v645/), [V727](https://pvs-studio.com/en/docs/warnings/v727/), [V755](https://pvs-studio.com/en/docs/warnings/v755/) |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Sanitize data supplied by users prior to use.  A user with bad intent could cause errors and security issues if they supply certain inputs (e.g., SQL injection). |

| **Noncompliant Code** |
| --- |
| This code example doesn’t check for appropriateness of user input prior to acting upon it. If the user provided input ending in “ or 1=1”, the SQL database will return all contents. |
| #include <iostream>  #include <string>  #include “sqlite3.h”  std::string userInput;  std::cin >> userInput;  sqlite3\_exec(database, userInput, callback, &output, &error\_message); |

| **Compliant Code** |
| --- |
| The following compliant code example uses a regular expression search to look for potential SQL injection in a user supplied string prior to acting upon it. If a SQL injection attempt is found, it prevents the SQL search and handles the attempt appropriately. |
| #include <iostream>  #include <string>  #include <regex>  #include “sqlite3.h”  std::string userInput;  std::cin >> userInput;  std::regex sqlInjectionSearch(“ [Oo][Rr] .\*=.\*”);  if (std::regex\_search(userInput, sqlInjectionSearch)) {  // SQL injection detected. Handle appropriately  } else {  sqlite3\_exec(database, userInput, callback, &output, &error\_message);  } |

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

| **Principles(s):**  1 – Validate that all data supplied by the user doesn’t contain anything malicious.  5 – Limit access to portions of the system that aren’t necessary for user activities. This can help to prevent security breaches if malicious code is supplied by the user.  7 – Clean all information provided by the user prior to sending it other systems (e.g., remove SQL injection prior to sending information to SQL database).  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | SQLI FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access previously freed memory.  Accessing memory that has previously been freed is undefined behavior and can result in security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant as it attempts to access a myClass object after it has been freed. |
| class myClass {  public:  void myFunction() {}  };  myClass \*myObject = new myClass;  delete myObject;  myObject->myFunction(); |

| **Compliant Code** |
| --- |
| The following compliant code example correctly accesses the myClass object prior to it being freed. |
| class myClass {  public:  void myFunction() {}  };  myClass \*myObject = new myClass;  myObject->myFunction();  delete myObject; |

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

| **Principles(s):**  3 – Design software to ensure that freed memory isn’t accessed.  8 – Stopping the code from attempting to access freed memory is just one layer in preventing security issues.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Detects duplicate memory deallocation or access of freed pointer. |
| Polyspace Bug Finder | R2024a | [CERT C++: MEM50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem50cpp.html) | Checks for duplicate memory deallocation or access of freed pointer. |
| PVS-Studio | 7.32 | [V586](https://pvs-studio.com/en/docs/warnings/v586/), [V774](https://pvs-studio.com/en/docs/warnings/v774/) |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions for diagnostic testing.  Assertions can be used as diagnostic testing to ensure that certain values are as expected. Care must be taken to ensure they are used correctly as a failed assert will cause immediate program termination. In many cases, the program should be allowed to gracefully terminate instead. Assertions are ideal during testing. |

| **Noncompliant Code** |
| --- |
| This code example uses assert to ensure that the constI variable wasn’t overwritten while obtaining user input (possibly from buffer overflow). The noncompliance comes from the fact that the assert function is left in the production code. If the constI variable is overwritten, the program will immediately terminate instead of handling the detected error. |
| // Code under development  #include <iostream>  char str[20];  const int constI = 100;  std::cin >> str;  assert(constI = 100);  // Production code  #include <iostream>  char str[20];  const int constI = 100;  std::cin >> str;  assert(constI = 100); |

| **Compliant Code** |
| --- |
| The following compliant code example uses the assert function during development but correctly updates this to another detection method in the production code which allows the program to handle the error instead of immediately terminating. |
| // Code under development  #include <iostream>  char str[20];  const int constI = 100;  std::cin >> str;  assert(constI = 100);  // Production code  #include <iostream>  char str[20];  const int constI = 100;  std::cin >> str;  if (constI != 100) {  // constI overwritten. Handle appropriately.  } |

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

| **Principles(s):**  9 – Assertions during development and debugging are great ways of testing that code is functioning as intended and adds to the quality assurance process.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.FUNCS.ASSERTS |  |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Checks for assertions that may have side effects. |
| Parasoft C/C++test | 2023.1 | CERT\_C-MSC11-a |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions.  Exceptions are thrown when issues are discovered during runtime and provide an opportunity to correct the issue if handled correctly. Failing to handle exceptions will cause the program to crash. |

| **Noncompliant Code** |
| --- |
| This code example throws an exception which is not handled and will cause the program to crash. |
| #include <stdexcept>  void throwException() {  throw std::system\_error(“Error”);  }  int main() {  throwException();  } |

| **Compliant Code** |
| --- |
| The following compliant code example uses try and catch blocks to handle thrown exceptions which prevents the program from crashing when an exception is encountered. |
| #include <stdexcept>  void throwException() {  throw std::system\_error(“Error”);  }  int main() {  try {  throwException();  } catch (…) {  \\ Exception encountered. Handle appropriately  }  } |

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

| **Principles(s):**  8 – Handling all exceptions will help to prevent crashes and can act as a line of defense against certain forms of attack.  9 – Unhandled exceptions will cause a program to crash. Preventing this will help to ensure quality for the end user.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Checks for unreachable catch statements. |
| Polyspace Bug Finder | R2024a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions. |
| RuleChecker | 22.10 | main-function-catch-all early-catch-all |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Files** | [STD-008-CPP] | Close files when no longer needed.  Failing to close files when they are no longer needed can lead to memory leaks. |

| **Noncompliant Code** |
| --- |
| This code example fails to close a file after it is no longer needed which can result in a loss of memory. |
| #include <fstream>  #include <string>  void myFunction(std::string fileName) {  std::fstream myFile(fileName);  // Perform tasks on the file  return;  } |

| **Compliant Code** |
| --- |
| The following compliant code example correctly closes the file when it is no longer needed. This prevents memory leaks. |
| #include <fstream>  #include <string>  void myFunction(std::string fileName) {  std::fstream myFile(fileName);  // Perform tasks on the file  file.close();  return;  } |

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

| **Principles(s):**  9 – Failure to close files when they are no longer needed can lead to a memory leak which will adversely affect the user’s experience.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | ALLOC.LEAK | Checks for memory leaks. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-FIO51-a | Checks that files are closed. |
| Parasoft Insure++ |  |  |  |
| Polyspace Bug Finder | R2024a | [CERT C++: FIO51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio51cpp.html) | Checks for memory leaks. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Object Initialization** | [STD-009-CPP] | Constructor member initializers should be in canonical order.  Member initializers are processed in the order that those member variables are declared. Incorrect declaration order can result in undefined behavior if the initialization of one depends on the initialization of another. |

| **Noncompliant Code** |
| --- |
| This code example produces undefined behavior due to the order in which the member variables are declared. The are initialized in the order declared. Due to this valueB isn’t initialized before valueA is which depends on the value of valueB. |
| class myClass {  private:  int valueA;  int valueB;  public:  myClass(int userInt) : valueB(userInt), valueA(valueB) {}  }; |

| **Compliant Code** |
| --- |
| The following compliant code example correctly declares the member variables in the correct order to initialization can proceed correctly. |
| class myClass {  private:  int valueB;  int valueA;  public:  myClass(int userInt) : valueB(userInt), valueA(valueB) {}  }; |

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

| **Principles(s):**  9 – Out of order initializers can cause undefined behavior. Preventing this will help to ensure quality.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP53 |  |
| CodeSonar | 8.1p0 | LANG.STRUCT.INIT.OOMI | Checks for member initializers that are out of order. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-OOP53-a | Checks for member initializers that are out of order. |
| Polyspace Bug Finder | R2024a | [CERT C++: OOP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcoop53cpp.html) | Checks for member initializers that are out of order. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Function Return Values** | [STD-010-CPP] | Functions that return a value must specify a return value from all possible exits.  Functions that have branching logic must provide a return value at all branches. Branches that don’t provide a return value introduce undefined behavior. |

| **Noncompliant Code** |
| --- |
| This code example can produce undefined behavior because not all logic branches provide a valid return value. Any supplied int other than 42 will not return a value. |
| bool myFunction(int myInt) {  if (myInt = 42) {  return true;  }  } |

| **Compliant Code** |
| --- |
| The following compliant code example correctly provides a return value for all logic branches and will not produce undefined behavior due to a missing return value. |
| bool myFunction(int myInt) {  if (myInt = 42) {  return true;  }  return false;  } |

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

| **Principles(s):**  2 – This error is likely to be caught by a complier or IDE. Reviewing complier warnings will help to prevent this error.  9 – Failure to provide return values at all possible exits presents the possibility of unexpected behavior which will reduce the quality of the final product.  10 – A secure coding standard can help prevent security risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC52 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MSC52-a | Checks that all exit paths have a return statement. |
| Polyspace Bug Finder | R2023a | [CERT C++: MSC52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmsc52cpp.html) | Checks for missing return statements. |
| PVS-Studio | 7.32 | [V591](https://pvs-studio.com/en/docs/warnings/v591/) |  |

### 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.

In the same way that DevOps is the intersection of development, IT operations, and application delivery, DevSecOps is the interaction of DevOps and security. All are equally important and all affect and influence the other. As such, using a DevSecOps requires that security be an equal focus from the beginning of the development process. As such, a focus on security will be injected into various steps in this process. For instance, the “Assess and plan”, “Verify and test”, “Monitor and detect”, and “Respond” steps all have a more manual component (e.g., predicting likely attack methods, testing for those attach methods, and monitoring for those attacks, respectively). But since automation is the intent in this review, focus will be given there.

Automation can factor heavily into the “Build” step. There are many automation tools (e.g., Cppcheck) that can review code for likely security vulnerabilities. This works similar to how a complier will review code and produce errors and warnings for syntax errors, but in much greater detail and will capture issues beyond what a complier can typically find. Additionally, some of these tools can be incorporated into IDEs with the intention of further easing the automation practice. As an example, Cppcheck can be added to Visual Studio and setup to review code with each save. Utilizing this feature and help the automation process as developers will no longer be required to use a second tool and thus will be more likely to review and act upon the findings from Cppcheck in a timely manner. Utilizing these tools (especially those described above in the Automation tables under each Coding Standard) can help to ensure that these standards are adhered to.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Medium | High | Medium | 3 |
| STD-002-CPP | High | High | High | Medium | 4 |
| STD-003-CPP | High | High | Medium | High | 5 |
| STD-004-CPP | High | High | Medium | High | 5 |
| STD-005-CPP | High | High | Medium | High | 5 |
| STD-006-CPP | Low | Low | High | Low | 1 |
| STD-007-CPP | Low | Medium | Medium | Medium | 2 |
| STD-008-CPP | Medium | Low | Medium | Medium | 2 |
| STD-009-CPP | Medium | Low | Medium | Medium | 2 |
| STD-010-CPP | Medium | High | Medium | Medium | 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 | Encryption at rest is the concept of keeping stored data in an encrypted format. This severely limits the ability of a hacker to steal and access stored information. To fully implement this concept, all stored data should be encrypted using a symmetric encryption scheme. Additionally, the keys used to decrypt that data will be stored separately. |
| Encryption in flight | Encryption in flight is the concept of encrypting all data that is transferred through a network. Doing so prevents the data from being intercepted in transit. To implement this practice, all data in flight should be encrypted using an asymmetric encryption scheme or TLS/SSL encryption. Both are ideal for data in transit as they allow anyone to encrypt data but only allow the intended recipient to decrypt that data. |
| Encryption in use | Encryption in use is the concept of keeping data encrypted while it is being acted upon. This further protects data as it doesn’t need to be decrypted to be worked on thereby preventing exposure to bad actors. To accomplish this, data in use should be encrypted using the Fast Fully Homomorphic Encryption over the Torus (TFHE) or the Simple Encrypted Arithmetic Library (Microsoft SEAL) schemes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of determining that a user is who they claim to be. This is typically done through a login process in which a user proves their identity through a unique username and password (ideally) known only to them. Additional authentication methods can be used (e.g., two factor authentication) depending on the level security needed. Failure to correctly authenticate a user can allow access to secure systems to an individual who should not have that access. To implement this, all user access to systems should require a username and password at a minimum and may require the use of additional security measures. |
| Authorization | Authorization is the process of granting specific access rights to authenticated users. By default, users should be granted the bare minimum access rights because failure to do so can lead to a situation in which a user has the ability to access and make changes to data and systems that they should not. Examples of various levels access can include the ability to read from and write to secure databases, add new users to a system, adjust access rights of other users, etc. To implement this, well defined access roles (with associated access rights) should be defined, and all users should be assigned to the one that provides the bare minimum level of access to perform their duties. As implied above, all new users should be assigned with no access rights until they can be assigned according to need. |
| Accounting | Accounting is the process of tracking and reviewing user activity within a secured system. The purpose of this can range from simple auditing to track which resources are being used and that hardware is able to keep up with demand on the low end all the way up to and including tracking user activity to prevent security risks and inappropriate access of files and resources by specific users. This is a necessary step to ensure that systems are functioning as intended and can act as part of the “Monitor and detect” step of the development process. |

**\***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 |  |
| 1.5 | 07/20/2024 | Coding Standards Created | Quintin B. Rozelle | [Insert text.] |
| 2.0 | 08/09/2024 | Completion | Quintin B. Rozelle | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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