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



# Green Pace Secure Development Policy

# Randolph Evanshill

## 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 validation is an important security measure that can help prevent vulnerabilities. (Injection attacks, buffer overflows, and other attacks that exploit malicious input data. By validating the data before it is processed by the program, vulnerabilities can be detected and prevented before exploitation. |
| 1. Heed Compiler Warnings | Heeding compiler warnings is an important security measure that can help to prevent issues contained within the code. The compiler warnings are generated by the compiler and indicate potential issues that may be in the code. If you fail to heed the warnings the code is vulnerable. |
| 1. Architect and Design for Security Policies | Architect and design for security policies entail designing the software with security in mind, from the earliest stages of development. This includes the design of secure software architectures, implementing secure coding practices, and performing threat modeling and risk assessments. |
| 1. Keep It Simple | The principle of keeping it simple involves designing software that is simple and easily understood. The purpose of this is that the more simple a program is the less likely the software is to contain vulnerabilities. What this does is makes it easier to keep secure and manage. |
| 1. Default Deny | The principle of default denial involves denying access to all resources by default. This is set up to only allow access to those that are explicitly required. |
| 1. Adhere to the Principle of Least Privilege | Adhering TO the principle of least privilege involves limiting users privileges to the minimum necessary so that they are able to still perform their tasks. This enables the reduction of risk of damage that may be caused if there is a vulnerability in the system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data that is sent to other systems involves validating data inputs to ensure their safety. This involves ensuring they are free of vulnerabilities and exploits and is a principle that can help to prevent unauthorized access or attacks. |
| 1. Practice Defense in Depth | Practicing defense in depth involves the implementation of multiple layers of security to protect against vulnerabilities. Each layer is provided to cover what the other layer may be lacking and the overlapping redundancy can create a stronger system. |
| 1. Use Effective Quality Assurance Techniques | Using effective quality assurance techniques ensures that the software that is designed meets the expected quality standards. Various techniques involved testing and validation to ensure that the software is free of defects and operates as intended. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard ensures that the software is developed with security in mind from the beginning of the project. It ensures that everyone involved is implementing coding guidelines that use the best practices for secure coding. |

### 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** | **Include the appropriate type information in function declarators** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Function declarators must be declared with the appropriate type information, including a return type and parameter list. If type information is not properly specified in a function declarator, the compiler cannot properly check function type information. When using standard library calls, the easiest (and preferred) way to obtain function declarators with appropriate type information is to include the appropriate header file. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the identifier-list form for parameter declarations: |
| int max(a, b)  int a, b;  {  return a > b ? a : b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, int is the type specifier, max(int a, int b) is the function declarator, and the block within the curly braces is the function body: |
| int max(int a, int b) {  return a > b ? a : b;  } |

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

| **Principles(s):** Consistency: It's important to maintain consistency throughout the codebase, especially when it comes to data types. The same data type should be used consistently across all functions and modules, and any conversions or casting should be done explicitly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not ignore values returned by functions** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Many functions return useful values whether or not the function has side effects. In most cases, this value is used to signify whether the function successfully completed its task or if some error occurred (see ERR02-C. Avoid in-band error indicators). Other times, the value is the result of some computation and is an integral part of the function's API.  Subclause 6.8.3 of the C Standard [ISO/IEC 9899:2011] states:  The expression in an expression statement is evaluated as a void expression for its side effects.  All expression statements, such as function calls with an ignored value, are implicitly cast to void. Because a return value often contains important information about possible errors, it should always be checked; otherwise, the cast should be made explicit to signify programmer intent. If a function returns no meaningful value, it should be declared with return type void.  This recommendation encompasses ERR33-C. Detect and handle standard library errors. Unlike this recommendation, that rule is restricted to functions from the Standard C library.  Compliance with this recommendation is required in order to comply with ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy |

| **Noncompliant Code** |
| --- |
| This noncompliant code example calls fputs() and fails to check whether a write error occurs: |
| FILE\* f = /\* output file stream \*/  /\* ... \*/  fputs("foo", f); |

| **Compliant Code** |
| --- |
| This compliant solution checks to make sure no output error occurred (see ERR33-C. Detect and handle standard library errors). |
| FILE\* f = /\* output file stream \*/  /\* ... \*/  if (fputs("foo", f) == EOF) {  /\* Handle error \*/  } |

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

| **Principles(s):** Accuracy: Data values should be accurate and correct. They should be validated and verified to ensure that they are correct and meet the required standards. This is especially important for critical data, such as financial or medical data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See undefined behavior 33.) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior: |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** Length: Strings should be of the correct length, neither too long nor too short, based on the intended use. If a string is too long, it can cause performance issues and if it is too short, it can lead to data loss or truncation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted. |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or [sanitized](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-sanitize). Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve [validating](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-validation) the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {  \*cp = '\_';  } |

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

| **Principles(s):** Parameterized queries: Use parameterized queries instead of concatenating strings to create SQL queries. Parameterized queries use placeholders for user input, which are then bound to the query at runtime. This makes it impossible for an attacker to inject malicious SQL code into the query. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Beware of zero-length allocations** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | When the requested size is 0, the behavior of the memory allocation functions malloc(), calloc(), and realloc() is implementation-defined. Subclause 7.22.3 of the C Standard [ISO/IEC 9899:2011] states:  If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.  In addition, the amount of storage allocated by a successful call to the allocation function when 0 bytes was requested is unspecified. See unspecified behavior 41 in subclause J.1 of the C Standard.  In cases where the memory allocation functions return a non-null pointer, reading from or writing to the allocated memory area results in undefined behavior. Typically, the pointer refers to a zero-length block of memory consisting entirely of control structures. Overwriting these control structures damages the data structures used by the memory manager. |

| **Noncompliant Code** |
| --- |
| The result of calling malloc(0) to allocate 0 bytes is implementation-defined. In this example, a dynamic array of integers is allocated to store size elements. However, if size is 0, the call to malloc(size) may return a reference to a block of memory of size 0 instead of a null pointer. When (nonempty) data is copied to this location, a heap-buffer overflow occurs. |
| size\_t size;    /\* Initialize size, possibly by user-controlled input \*/    int \*list = (int \*)malloc(size);  if (list == NULL) {  /\* Handle allocation error \*/  }  else {  /\* Continue processing list \*/  } |

| **Compliant Code** |
| --- |
| To ensure that 0 is never passed as a size argument to malloc(), size is checked to confirm it has a positive value: |
| size\_t size;    /\* Initialize size, possibly by user-controlled input \*/    if (size == 0) {  /\* Handle error \*/  }  int \*list = (int \*)malloc(size);  if (list == NULL) {  /\* Handle allocation error \*/  }  /\* Continue processing list \*/ |

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

| **Principles(s):** Memory segmentation: Use memory segmentation to separate the user mode and kernel mode memory spaces. This can help prevent malicious code from accessing or modifying kernel memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | [Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see MSC11-C. Incorporate diagnostic tests using assertions). The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Do not let exceptions escape from destructors or deallocation functions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Under certain circumstances, terminating a destructor, operator delete, or operator delete[] by throwing an exception can trigger undefined behavior.  For instance, the C++ Standard, [basic.stc.dynamic.deallocation], paragraph 3 [ISO/IEC 14882-2014], in part, states the following:  If a deallocation function terminates by throwing an exception, the behavior is undefined.  In these situations, the function must logically be declared noexcept because throwing an exception from the function can never have well-defined behavior. The C++ Standard, [except.spec], paragraph 15, states the following:  A deallocation function with no explicit exception-specification is treated as if it were specified with noexcept(true).  As such, deallocation functions (object, array, and placement forms at either global or class scope) must not terminate by throwing an exception. Do not declare such functions to be noexcept(false). However, it is acceptable to rely on the implicit noexcept(true) specification or declare noexcept explicitly on the function signature. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the class destructor does not meet the implicit noexcept guarantee because it may throw an exception even if it was called as the result of an exception being thrown. Consequently, it is declared as noexcept(false) but still can trigger undefined behavior. |
| #include <stdexcept>    class S {  bool has\_error() const;    public:  ~S() noexcept(false) {  // Normal processing  if (has\_error()) {  throw std::logic\_error("Something bad");  }  }  }; |

| **Compliant Code** |
| --- |
| A destructor should perform the same way whether or not there is an active exception. Typically, this means that it should invoke only operations that do not throw exceptions, or it should handle all exceptions and not rethrow them (even implicitly). This compliant solution differs from the previous noncompliant code example by having an explicit return statement in the SomeClass destructor. This statement prevents control from reaching the end of the exception handler. Consequently, this handler will catch the exception thrown by Bad::~Bad() when bad\_member is destroyed. It will also catch any exceptions thrown within the compound statement of the function-try-block, but the SomeClass destructor will not terminate by throwing an exception. |
| class SomeClass {  Bad bad\_member;  public:  ~SomeClass()  try {  // ...  } catch(...) {  // Catch exceptions thrown from noncompliant destructors of  // member objects or base class subobjects.    // NOTE: Flowing off the end of a destructor function-try-block causes  // the caught exception to be implicitly rethrown, but an explicit  // return statement will prevent that from happening.  return;  }  }; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Choose an appropriate termination strategy** |
| --- | --- | --- |
| Error Handling | [STD-008-CPP] | Some errors, such as out-of-range values, might be the result of erroneous user input. Interactive programs typically handle such errors by rejecting the input and prompting the user for an acceptable value. Servers reject invalid user input by indicating an error to the client while at the same time continuing to service other clients' valid requests. All robust programs must be prepared to gracefully handle resource exhaustion, such as low memory or disk space conditions, at a minimum by preventing the loss of user data kept in volatile storage. Interactive programs may give the user the option to save data on an alternative medium, whereas network servers may respond by reducing throughput or otherwise degrading the quality of service. However, when certain kinds of errors are detected, such as irrecoverable logic errors, rather than risk data corruption by continuing to execute in an indeterminate state, the appropriate strategy may be for the system to quickly shut down, allowing the operator to start it afresh in a determinate state. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] The abort() function should not be called if it is important to perform application-specific cleanup before exiting. In this noncompliant code example, abort() is called after data is sent to an open file descriptor. The data may or may not be written to the file. |
| #include <stdlib.h>  #include <stdio.h>    int write\_data(void) {  const char \*filename = "hello.txt";  FILE \*f = fopen(filename, "w");  if (f == NULL) {  /\* Handle error \*/  }  fprintf(f, "Hello, World\n");  /\* ... \*/  abort(); /\* Oops! Data might not be written! \*/  /\* ... \*/  return 0;  }    int main(void) {  write\_data();  return EXIT\_SUCCESS;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to abort() is replaced with exit(), which guarantees that buffered I/O data is flushed to the file descriptor and the file descriptor is properly closed: |
| #include <stdlib.h>  #include <stdio.h>    int write\_data(void) {  const char \*filename = "hello.txt";  FILE \*f = fopen(filename, "w");  if (f == NULL) {  /\* Handle error \*/  }  fprintf(f, "Hello, World\n");  /\* ... \*/  exit(EXIT\_FAILURE); /\* Writes data and closes f \*/  /\* ... \*/  return 0;  }    int main(void) {  write\_data();  return EXIT\_SUCCESS;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logic | [STD-009-CPP] | [Rationalize the standard.] |

| **Noncompliant Code** |
| --- |
| This noncompliant code example fails to test for conditions where a is neither b nor c. This behavior may be correct in this case, but failure to account for all the values of a can result in logic errors if a unexpectedly assumes a different value. |
| if (a == b) {  /\* ... \*/  }  else if (a == c) {  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution explicitly checks for the unexpected condition and handles it appropriately: |
| if (a == b) {  /\* ... \*/  }  else if (a == c) {  /\* ... \*/  }  else {  /\* Handle error condition \*/  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not invoke virtual functions from constructors or destructors** |
| --- | --- | --- |
| Object Oriented Programming | [STD-010-CPP] | Virtual functions allow for the choice of member function calls to be determined at run time based on the dynamic type of the object that the member function is being called on. This convention supports object-oriented programming practices commonly associated with object inheritance and function overriding. When calling a nonvirtual member function or when using a class member access expression to denote a call, the specified function is called. Otherwise, a virtual function call is made to the final overrider in the dynamic type of the object expression. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| struct B {  B() { seize(); }  virtual ~B() { release(); }    protected:  virtual void seize();  virtual void release();  };    struct D : B {  virtual ~D() = default;    protected:  void seize() override {  B::seize();  // Get derived resources...  }    void release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources. |
| class B {  void seize\_mine();  void release\_mine();    public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }    protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };    class D : public B {  void seize\_mine();  void release\_mine();    public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }    protected:  void seize() override {  B::seize();  seize\_mine();  }    void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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.] |

### 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.

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| [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.] |
| [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.] |
| [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.] |
| [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 in rest | Encryption in rest refers to the process of converting plain text data or information into an unreadable format using an algorithm. This should be put in place to protect sensitive information and prevent unauthorized access. |
| Encryption at flight | This refers to the process of protecting data that is in the process of being transmitted between systems or sent over a network. |
| Encryption in use | Encryption in use is the protection of data while it is being stored by a system or application. This is done to ensure that data is protected even if the underlying storage device or system becomes compromised. This involves encrypting data in real-time as it is being used by a system or application. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user, system, or device attempting to access a resource or service. It involves confirming that the identity claimed by the user or system is legitimate and authorized to access the resource or service. |
| Authorization | Authorization is the process of determining whether a user, system, or device has the necessary permissions and privileges to access a resource or perform an action. It involves checking the access rights and privileges of the user or system to ensure that they are authorized to perform the requested action or access the requested resource. |
| Accounting | Accounting involves the collection, analysis, and reporting of security-related events and activities, such as login attempts, file accesses, system changes, and other security-relevant events. This information can be used to detect and respond to security incidents, track compliance with security policies, and provide audit trails for forensic investigations. |

**\***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 |