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

Project Two: Security Policy Presentation

Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

# Overview

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

# Purpose

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

# Scope

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

# Module Three Milestone

## Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data, or simply input validation, is a highly important concept when it comes to secure coding. Any time a program is receiving input across some means of communication not directly contained within the program, input validation must be performed. Some examples times where input validation needs to be performed include environment variables, data read from sockets, pipes, files, signals, shared memory, and devices. |
| 1. Heed Compiler Warnings | While compiler warnings to not prevent your code from compiling, heeding the warnings and fixing the code which they warn you about is a low effort way to avoid some potentially hard to find bugs. One example that might be more difficult for a developer to notice but it easy for the compiler to notice and emit a warning for is a case of using a single ‘=’ versus a double ‘==’ when evaluating a Boolean condition. A compiler will emit a warning for this condition and the developer could choose to ignore this condition because the code will compile; however, because it will always be true, it may lead to serious issues with the program. |
| 1. Architect and Design for Security Policies | Company and business security policies must always be considered when architecting and designing new systems. Typically, these policies are enacted by system and network administrators but must be upheld at the source code level to maintain consistency. |
| 1. Keep It Simple | Do not overcomplicate security solutions to the point of convoluting code. For instance, it is a good idea to maintain a Defense-in-Depth strategy when implementing solutions, but it is important to keep in the mind the limitations that go along with the strategy. In other words, do not code more than you need to. |
| 1. Default Deny | When implementing classes, methods, or functions that interact with external connections for means of sharing data, a default deny policy must be implemented. This can be thought of as a “whitelist” approach, where any connection that is not explicitly allowed and supported should not be trusted, thus denied the ability to connect. |
| 1. Adhere to the Principle of Least Privilege | Any time a user related operation is implemented, always consider the principle of least privilege. Authorization classes and operations should be implemented in such a way that a user cannot access that part of the application. Also, a user should have access only to the minimum privileges absolutely necessary to perform their intended function. |
| 1. Sanitize Data Sent to Other Systems | If you are working with any data within your code that needs to be sent to an external system (database, external server, etc.) any traces of the data must be sanitized. |
| 1. Practice Defense in Depth | As previously mentioned, practicing Defense-in-Depth strategy is an effective way to minimize security risks during development. For example, when working with strings it may be unnecessary for a developer to use bounded string copy operations from community resources if all strings go through input validation. Doing so could be considered “too deep” within DiD and could lead to inefficiencies with speed or memory usage within the application. |
| 1. Use Effective Quality Assurance Techniques | Any application code introduced should go through the same quality assurance (QA) techniques. Some of the standard QA procedures that should take place during development include but are not limited to unit tests, integration tests, and pull request. These three strategies are a starting point for quality assurance techniques. |
| 1. Adopt a Secure Coding Standard | A secure coding standard will ensure all developers within the organization or team will adhere to the same level of security. A coding standard can limit the amount of ambiguity surrounding which security measures must be implemented during application development. It can also provide the developers with a reminder of security practices to implement when they are in development. |

## 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] | Do not qualify a reference type with const |

| **Noncompliant Code** |
| --- |
| When attempting to const-qualify a type as part of a declaration that uses reference type, a programmer may accidentally write the following code: |
| **char** &**const** p; |

| **Compliant Code** |
| --- |
| Due to the fact that C++ treats all reference types as const, the correct way of writing this code would be: |
| **char** **const** &p; // Or: const char &p; |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings: Depending on the compiler and version being used, the placement of the reference symbol, ‘&’, may be presented during compilation as a warning. It is important for the developer to heed this warning and take the action necessary to ensure the appropriate reference symbol placement. |

**Threat Level**

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

**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** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Check for wrapping when using unsigned integers. |

| **Noncompliant Code** |
| --- |
| The following code assumes the values in ui\_a and ui\_b add up to a number that does not exceed the maximum value for an unsigned integer: |
| |  | | --- | | **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {  unsigned **int** usum = ui\_a + ui\_b;  /\* ... \*/  } | |

| **Compliant Code** |
| --- |
| Use standard library constants to check sum values before assigning them to unsigned integers: |
| #include <limits.h>    **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {  unsigned **int** usum;  **if** (UINT\_MAX - ui\_a < ui\_b) {  /\* Handle error \*/  } **else** {  usum = ui\_a + ui\_b;  }  /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** ValidateInput Data: Many wrapping errors involving numerical data types can be caught before damage is done within the application. This is best caught at runtime by validating the input data. Another method would be to implement proper quality assurance techniques to catch wrapping errors. |

**Threat Level**

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

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-LLL] | Check for buffer overflow in string copy operations. |

| **Noncompliant Code** |
| --- |
| The following code example does not account for the null terminator when iterating over the char array: |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n); ++i) {  dest[i] = src[i];  }  dest[i] = '\0';  } |

| **Compliant Code** |
| --- |
| The correct way to iterate over a char array and to avoid buffer overflow would be the following: |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n - 1); ++i) {  dest[i] = src[i];  }  dest[i] = '\0';  } |

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

|  |
| --- |
| **Principles(s):** Practice Defense in Depth: String copy operations are a common C++ operation that benefit heavily from Defense-in-Depth (DiD). There are many well established string libraries, including some in the standard library, that include useful, highly tested string copy operations. The best approach is a DiD approach: use well tested code to handle common operations, and perform runtime validation for multiple layers of protection. |

**Threat Level**

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

**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** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Perform proper validation of ORs when executing SQL to avoid SQL injection. |

| **Noncompliant Code** |
| --- |
| This code runs a statement against SQLlite without any validation: |
| #include <vector>  #include "sqlite3.h"  bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  /\* … \*/  char\* error\_message;  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  /\* … \*/  return false;  }  return true;  } |

| **Compliant Code** |
| --- |
| This code, with the help of a utility function, checks if two operands of an OR clause are equal before executing the SQL: |
| #include <vector>  #include "sqlite3.h"  bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  /\* … \*/  char\* error\_message;  lhs = right\_operand(sql); // some utility function to get operands  rhs = right\_operand(sql);  if (lhs == rhs)  { // SQL injection!!  std::cout << "Possible SQL injection detected, ignoring query: '" << sql << "'" << std::endl;  return true;  }  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  /\* … \*/  return false;  }  return true;  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques: Effective Quality Assurance Techniques will catch areas of the code that do not protect against SQL injection. Due to the dangerous nature of SQL injection, a DiD strategy is also recommended to defend against this vulnerability, though it is important to start with effective QA techniques. |

**Threat Level**

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

**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** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Initialize all pointers that are not class members as smart pointers. |

| **Noncompliant Code** |
| --- |
| The following code example creates a raw pointer to an object and does not free the memory, leading to a memory leak within the application: |
| void func() {  Object\* obj = new Object();  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The safest way of allocating an object on the heap in C++ is to utilize smart pointers: |
| void func() {  Object\* obj = new Object();  std::unique\_ptr<Object> u\_obj(obj);  /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** Keep It Simple: Because more modern versions of C++ offer tools such as smart pointers, it may be an effective strategy to require smart pointer usage as part of a coding standard. This simplifies the allocation of memory in all aspects of the application. |

**Threat Level**

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

**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** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Proper usage of assert() statements. |

| **Noncompliant Code** |
| --- |
| This code block uses assert() improperly, preventing the exit functions for performing: |
| void cleanup(void) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(void) {    if (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **assert**(/\* Something bad didn't happen \*/);      /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Instead, the proper usage for this code would be to call exit() with some constant for the exit status, in this case, failure: |
| void cleanup(void) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(void) {    if (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/      if (/\* Something bad happened \*/) {  **exit**(EXIT\_FAILURE);    }      /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques: Because assertions are used to prevent more obvious errors in code, proper usage of assert() statements falls under the effective QA principle. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | Low | 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 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle leaks along with exception handling. |

| **Noncompliant Code** |
| --- |
| The following example does not handle the memory *pst* if an exception in thrown: |
| #include <new>    struct SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.    void process\_item() noexcept(false);  };    void f() {    SomeType \*pst = new (std::nothrow) SomeType();    if (!pst) {      // Handle error      return;    }      try {      pst->process\_item();    } catch (...) {      // Process error, but do not recover from it; rethrow.      throw;    }    delete pst;  } |

| **Compliant Code** |
| --- |
| The following code properly handles the leak in this case: |
| #include <new>    struct SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.      void process\_item() noexcept(false);  };    void f() {    SomeType \*pst = new (std::nothrow) SomeType();    if (!pst) {      // Handle error      return;    }    try {      pst->process\_item();    } catch (...) {      // Process error, but do not recover from it; rethrow.      delete pst;      throw;    }    delete pst;  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard: A combination of a secure coding standard (e.g. requiring the use of smart pointers for all allocated memory) and QA techniques are the most relevant principles upheld with this standard. |

**Threat Level**

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

**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** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-008-CPP] | Properly handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| The following function is marked with *noexcept* even though the allocation of an integer array can throw an exception: |
| #include <cstring>    void f(const **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = new **int**[size];    std::**memcpy**(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

| **Compliant Code** |
| --- |
| Instead, use *std::nothrow* and check that the pointer is not null before continuing: |
| #include <cstring>  #include <new>    void f(const **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = new (std::nothrow) **int**[size];    if (!copy) {      // Handle error      return;    }    std::**memcpy**(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard: This standard is part of an overall secure coding standard. It is important to uphold to ensure invalid memory is not referenced within the application. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | Low | 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 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input and Output** | [STD-009-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| The following example opens an *fstream* object and does not close the object before calling *std::terminate*: |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| Any *fstream* object that is opened must be properly closed within the same scope: |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

|  |
| --- |
| **Principles(s):** Sanitize Data Sent to Other Systems: This standard adheres to the Sanitizing Data principle. Because file systems are systems external to the application, it is important to remove all traces of the file from within the application. This, of course, includes closing *fstreams* when the application processing has completed. |

**Threat Level**

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

**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** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-010-CPP] | Do not attempt to create a string from a nullptr. |

| **Noncompliant Code** |
| --- |
| The following code uses *std::getenv* to assign a value to a *std::string* object. This function can return a nullptr and result in undefined behavior: |
| #include <cstdlib>  #include <string>    void f() {    std::string tmp(std::**getenv**("TMP"));    if (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The proper approach is to check for a null value before assigning it to a string: |
| #include <cstdlib>  #include <string>    void f() {    const **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");    if (!tmp.empty()) {      // ...    }  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard: Adopting a secure standard would alleviate this potential security violation. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Low | 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.] |

## 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.

Many of the security principles defined in the security policy document do not apply directly to the DevOps process; however, this does not mean there are not important principles defined that should be adopted. For example, **Architecting and Designing** for security principles MUST be applied in the DevSecOps process. More specifically, to comply with the principles defined in this document, those involved in the DevSecOps process must ensure that “Build”, “Design”, and “Verify and test” steps shown in the “Pre-production” section within the DevSecOps diagram above need to uphold the security policies and standards defined in this document. This will include implementing proper **Quality Assurance** techniques for DevSecOps process, during Pre-production and production (e.g. always running unit tests when performing application builds). It is important to note that in order to uphold the level of standard expected by this document, ALL members involved with the production and deployment of the application MUST adhere to the standards defined.

## 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 | Low | 1 |
| STD-002-CPP | Medium | Likely | Medium | Low | 2 |
| STD-003-CPP | High | Likely | High | High | 4 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | Medium | Likely | Low | Low | 1 |
| STD-006-CPP | Medium | Unlikely | Low | Low | 1 |
| STD-007-CPP | Medium | Likely | Low | Medium | 3 |
| STD-008-CPP | Low | Likely | Medium | Low | 2 |
| STD-009-CPP | Medium | Unlikely | Medium | Medium | 3 |
| STD-010-CPP | Low | Likely | Low | Low | 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.] |

## 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 | This policy involves securing any data related to the application that will not be accessed by the application at high volumes. For example, sensitive customer data that does not have a high read frequency, like invoices from customer orders, should be encrypted at rest. This type of encryption may also apply to file encryption on application servers.  For low-read-frequency, application data, AES-128 encryption using a company password is likely a sound approach. It is important that only authorized personnel are aware of this password and DO NOT share it with anyone else, including other members of the team. |
| Encryption in flight | This policy involves the transmission of data from one system to another and ensuring that said data is appropriately encrypted. To uphold this policy, all communication performed over application layer protocols must be done so via a secure protocol. For example, web-servers providing data to clients should only do so via HTTPS. RSA keys and X509 certificates used to establish TLS must never be shared outside authorized members within the team. |
| Encryption in use | This policy refers to data active within an application’s runtime. This policy is important to uphold so that malicious users do not have the ability to garner sensitive information or data from an application in use. One practical example might be hashes performed on data for validation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the first step of the Triple-A framework. A client should not be provided access to internal resources without the proper credentials. |
| Authorization | Once a user has entered the correct credentials and has been granted access to internal resources, the next step of the Triple-A framework is to provide that user with authorization. This step provides the user their level of access within the application along with which operations they are allowed to perform. |
| Accounting | The third step of the Triple-A framework. This step is typically not visible to the user and is mostly for monitoring purposes. This involves session management and application logs that determine which operations were performed by which level of authorized user. Accounting is highly important and should not be overlooked when implementing the Triple-A framework. |

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

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

## Map the Principles

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

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

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

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

# Audit Controls and Management

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

Evidence will include the following:

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

# Enforcement

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

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

# Exceptions Process

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

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

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

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

# Distribution

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

# Policy Change Control

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

# Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 08/08/2021 | Added coding standards and completed risk assessment. | Jesse Quijano |  |
| [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 |