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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 12](#_Toc52464063)

[Coding Standard 6 14](#_Toc52464064)

[Coding Standard 7 16](#_Toc52464065)

[Coding Standard 8 18](#_Toc52464066)

[Coding Standard 9 20](#_Toc52464067)

[Coding Standard 10 22](#_Toc52464068)

[Defense-in-Depth Illustration 24](#_Toc52464069)

[Project One 24](#_Toc52464070)

[1. Revise the C/C++ Standards 24](#_Toc52464071)

[2. Risk Assessment 24](#_Toc52464072)

[3. Automated Detection 24](#_Toc52464073)

[4. Automation 24](#_Toc52464074)

[5. Summary of Risk Assessments 25](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 25](#_Toc52464076)

[7. Map the Principles 26](#_Toc52464077)

[Audit Controls and Management 27](#_Toc52464078)

[Enforcement 27](#_Toc52464079)

[Exceptions Process 27](#_Toc52464080)

[Distribution 28](#_Toc52464081)

[Policy Change Control 28](#_Toc52464082)

[Policy Version History 28](#_Toc52464083)

[Appendix A Lookups 28](#_Toc52464084)

[Approved C/C++ Language Acronyms 28](#_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

Source: <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate all input to prevent vulnerabilities. Assume that all external data sources are unsafe, i.e., user-controlled files. |
| 1. Heed Compiler Warnings | All compiler warnings must be addressed. Remove compiler warnings by modifying code and through testing. Use testing to detect and remove additional security risks. |
| 1. Architect and Design for Security Policies | Design the software to adhere to security policy. Build the software in using the design. Do not get sloppy, stay focused. |
| 1. Keep It Simple | Keep it simple, code in small chunks. Don’t get fancy. |
| 1. Default Deny | Permission based access, not exclusion. No credentials, no access! |
| 1. Adhere to the Principle of Least Privilege | For all users, processes, programs, and networks, each process will run with the least amount of privilege needed. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data before moving it along to another system. |
| 1. Practice Defense in Depth | Protect system by layering defenses. Multiple strategies in place to help prevent vulnerabilities and respond to any exploits. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance will help to identify and eliminate vulnerabilities and exploits. |
| 1. Adopt a Secure Coding Standard | Adopt a secure coding standard with each programming language being used. Be aware of common vulnerabilities associated with each language, |

### 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

Source: <https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CLG] | Do not allow implicit declarations. Declare variables, functions, and returns. |

| **Noncompliant Code** |
| --- |
| Implicit variable |
| foo = 0; |

| **Compliant Code** |
| --- |
| Declared variable |
| int foo = 0; |

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

| **Principle(s):**  1 – Defined data types must match Input.  2 – Correct all compiler warnings related to data type declarations.  3 – Explicit variable declarations are easier to track and understand than implicit declarations.  4 – For an extra layer of defense, use Variable, Function, and Return declarations. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Type-specifier  Function-return-type  Implicit-function-declaration  Undeclared-parameter | Fully checked |
| ECLAIR | 1.2 | CERTC-DCL31 | Fully Implemented |
| Parasoft  C/C++test | 2021.2 | CERTC-DCL31 | All functions shall be declared before  use. |
| Polyspace Bug  Finder | R2021a | CERTC-DCL31 | Types not explicitly specified;  Implicit function declaration. |

#### Coding Standard 2

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not read uninitialized memory. |

| **Noncompliant Code** |
| --- |
| Uninitialized local variable is evaluated as part of an expression to print its value. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| Object is initialized prior to printing its value |
| #include <iostream>  void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):**  1 – Input should be within the ranges defined by variable initialization.  2 – Correct all compiler warnings related to uninitialized variables.  3 – Initializing variables will add a layer of defense. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | PROBABLE | MEDIUM | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Uninitialized-read | Partially checked |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Polyspace Bug  Finder | R2021a | CERT C++;EXP53-CPP | Non-initialized variable;  Non-Initialized pointer;  Rule partially covered. |
| Parasoft Insure++ |  |  | Runtime detection |

#### Coding Standard 3

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee storage for strings has sufficient space for character data and null terminator. |

| **Noncompliant Code** |
| --- |
| Unbounded input |
| #include <iostream>  void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Use std::string to guard against buffer overflows and data truncation. |
| #include <iostream>  #include <string>  void f() {  std::string input;  std::string stringOne, stringTwo:  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):**  1 – Storage for input string must allow for Null terminator space.  7 – String lengths bounded by Null terminator prevent occurrences of SQL injection and buffer overruns.  8 – String length verification provides a layer of defense against vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer Overrun  Type overrun |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-STR5-b  CERT\_CPP-STR5-c  CERT\_CPP-STR5-e  CERT\_CPP-STR5-f  CERT\_CPP-STR5-g | Avoid overflow due to reading a not zero terminated string,  Avoid overflow when writing o a buffer,  Prevent buffer overflows from tainted data,  Avoid buffer write overflow from tainted data,  Do not use the ‘char’ buffer to store input from `std::cin` |
| Polyspace Bug Finder | R2021b | CERT C++;STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation   Rule partially covered |

#### Coding Standard 4

Source:

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CLG] | Use prepared statements for queries. |

| **Noncompliant Code** |
| --- |
| Without prepared statements users can alter the SQL statements. |
| uName = getRequestString(“username”);  uPass = getRequestString(“userpassword”);  sql = “SELECT \* FROM Users WHERE Name = “ + uName + “ AND Pass = “ + uPass + “ |

| **Compliant Code** |
| --- |
| User does not have control of the SQL statement when using prepared statements. This will limit potential injections. |
| PreparedStatement pStmt = PreparedStatement();  std::cin >> username;  std::cin >> userpassword;  sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username,  userpassword}; |

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

| **Principles(s):**  1 – Validate input to prevent SQL injection.  3 – Use prepared SQL statements to prevent injection.  7 – Usage of prepare statements allow data to be sanitized as they add a layer of defense against SQL injection attacks |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.2 | FIO30-c | Never use unfiltered data from an untrusted user as the format parameter |
| Astre’e | 20.10 |  | Supported via stubbing/taint analysis |

#### Coding Standard 5

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| s is dereferenced after it has been deallocated, resulting in a write-after-free. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| Dynamically allocated memory is not deallocated until it is no longer needed. |
| #include <new>  **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |

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

| **Principles(s):**  2 – Use static tools to detect, identify, and mitigate freed memory issues.  9 – Use effective QA techniques to identify freed memory occurrences that could be easily overlooked. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Dangling\_pointer\_use |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Polyspace Bug Finder | R2021b | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered |

#### Coding Standard 6

Source:

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to test assumptions. If the expression is false, abort() is called preventing unexpected behavior. |

| **Noncompliant Code** |
| --- |
| Here a pointer may be NULL. This can result in a exploitable vulnerability. |
| #include <stdio.h>  #include <assert.h>  void print\_number(int\* myInt) {  printf (“%d\n”, \*myInt);  }  int main ()  {  int a = 10;  int \* b = NULL;  int \* c = NULL;  b=&a;  print\_number (b);  print\_number (c);  return 0;  } |

| **Compliant Code** |
| --- |
| Use assert() to prevent printing from dereferenced memory. |
| #include <stdio.h>  #include <assert.h>  void print\_number(int\* myInt) {  assert (myInt!=NULL);  printf ("%d\n",\*myInt);  }  int main ()  {  int a = 10;  int \* b = NULL;  int \* c = NULL;  b=&a;  print\_number (b);  print\_number (c);  return 0;  } |

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

| **Principles(s):**  9 – Use assertions to test code throughout the program. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions  Do not leave ‘catch’ blocks empty |

#### Coding Standard 7

Source:

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| No handler in either f() or main() for the exception thrown in throwing\_func(), calling on std::terminate(). |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| Use a try/catch block in main() to handle exceptions. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  try {  f();  } catch (...) {  // error  }  } |

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

| **Principles(s):**  3 – Design code using try, catch, and throw to prevent code from stopping suddenly.  9 – Continually test throughout code and throw exceptions to prevent unexpected behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | PROBABLE | MEDIUM | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | main-function-  catch-all  early-catch-all | Partially checked |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception thrown in the code shall have a handler of a compatible type in all call path that could lead to that point |
| Polyspace Bug Finder | R2021b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered |

#### Coding Standard 8

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | [STD-008-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| std::fstream object file is constructed, but it is not properly closed. |
| #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** |
| --- |
| std::fstream::close() is called before std::terminate() ensuring file resources are closed properly. |
| #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):**  8 – Closing open files provides a layer of defense, restricting unpermitted data access  10 – Closing open files is best practice |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | UNLIKELY | MEDIUM | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | ALLOC.LEAK | leak |
| Klocwork | 2022.1 | RH.LEAK |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-fIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2021b | CERT C++: FIO51-CPP | Checks for resource leak  Rule partially covered |

#### Coding Standard 9

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Use valid iterator ranges. |

| **Noncompliant Code** |
| --- |
| Undefined behavior resulting from incorrect values being passed improperly to the for\_each() function. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| The iterator values passed to std::for\_each() are passed in the proper order. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):**  3 – Protect iterators from generating overflow errors.  4 – Explicitly define iterator ranges. Makes code simple and understandable.  8 – Preventing iterator overflow adds a layer of defense. Also, best practice. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | PROBABLE | HIGH | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | overflow\_upon\_dereference |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn’t really a range  Do not compare iterators from different containers |

#### Coding Standard 10

Source: <https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-010-LLL] | Ensure that operations on signed integers do not result in overflow. |

| **Noncompliant Code** |
| --- |
| This example can result in a signed integer overflow. |
| **void** func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This example ensures the addition operation cannot overflow. |
| #include <limits.h>    **void** f(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum;  **if** (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||        ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {      /\* Handle error \*/    } **else** {      sum = si\_a + si\_b;    }    /\* ... \*/  } |

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

| **Principles(s):**  1 – Validate input that may cause overflow  9 – Always surround computations with tests for integer overflow and handle exceptions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | integer-overflow | Fully checked |
| Parasoft C/C++test | 2021.2 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid integer overflows  Integer overflow or underflow in constant expression in ‘+’, ‘-‘, ‘\*’ operator  Integer overflow or underflow in constant expression in ‘<<’ operator |
| Polyspace Bug Finder | R2021a | CERT C: RULE INT32-C | Checks for:   * Integer overflow * Tainted division operand * Tainted modulo operand   Rule partially covered. |

### 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-CLG | LOW | UNLIKELY | LOW | 3 | 3 |
| STD-002-CPP | HIGH | PROBABLE | MEDIUM | 12 | 1 |
| STD-003-CPP | HIGH | LIKELY | MEDIUM | 18 | 1 |
| STD-004-CLG | HIGH | LIKELY | MEDIUM | 18 | 1 |
| STD-005-CPP | HIGH | LIKELY | MEDIUM | 18 | 1 |
| STD-006-CPP | HIGH | LIKELY | HIGH | 9 | 2 |
| STD-007-CPP | LOW | PROBABLE | MEDIUM | 4 | 3 |
| STD-008-CPP | MEDIUM | UNLIKELY | MEDIUM | 4 | 3 |
| STD-009-CPP | HIGH | PROBABLE | HIGH | 6 | 2 |
| STD-010-CLG | HIGH | LIKELY | HIGH | 9 | 2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Protects data where it is stored, whether it is a physical hard drive or logical source like a database or cloud storage. There are many options for encryption tools, so finding one that will work for what is needed should not be a problem. These tools are necessary to protect data from being stolen. |
| Encryption at flight | Protects data while it is moving from one location to another, emails/internet. Email encryption tools will be used for email transmissions. Web traffic will be sent over Secure Sockets Layer (SSL) like Transport Layer Security (TLS) by obtaining a SSL/TLS certificate. Using these tools will prevent network layer attacks and third-party communication hijacks. |
| Encryption in use | Protects data while it is use, creating, editing, processing, etc. The in-use state is the state between in-rest and at-flight states. Encrypting data that is in-use is important because memory can be hacked and the encryption keys for in-rest can be exposed. Many companies offer full memory encryption in order to protect the CPU based key storage. You can also find cryptographic tools to protect data during computation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Who are you? Verifies a user’s identity credentials. Usually with a username/password combo, but other options are available. All users will be verified. Identity theft and unauthorized use can happen if the user’s credentials are not authenticated. |
| Authorization | Where can you go? Authorization is set for each user allowed on the system and allows the user access to only what is needed to complete each user’s task. Read, write, execution, administrator, etc. |
| Accounting | What happened? When did it happen? This refers to records and log files, which detail everything that is going on. Create and new user profile? Logged and timestamped! Accessed a file? Logged and stamped! Data transfer? Logged and stamped! User’s are being tracked at all times. |

**\***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 |  |
| 2.0 | 03-20-2022 | Milestone 3 | Ryan Biekert |  |
| 3.0 | 04-09-2022 | Project 1 | Ryan Biekert | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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