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

**6-2 Project One - Fernando Lomeli**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input data validation is the process of inspecting and filtering out the input data before it enters into the system. |
| 1. Heed Compiler Warnings | Compiler warnings warn the user about any possible problems within the code. Compiler warnings do not stop the code from compiling, but they are still essential for highlighting possible issues and leading to improved code quality. |
| 1. Architect and Design for Security Policies | Using proper architecture and design when it comes to designing, creating, and implementing security policies for the company. Prioritizing security in the design process is essential. |
| 1. Keep It Simple | Keep it simple is a design principle in which developers should keep the code as simple and efficient as possible. This ensures the code is less complex, easy to understand, and straightforward. |
| 1. Default Deny | Default deny is a policy in which access to secure data is denied by default unless the user is authorized to access the data. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege enables only the required permissions (no more and no less) for a user to complete their task. This ensures a good balance of usability and security. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before it is sent to other systems is essential in preventing attacks like SQL injection. Removing or sanitizing certain characters that can trigger certain commands or actions that compromise the system is essential to preventing this vulnerability. |
| 1. Practice Defense in Depth | Defense in depth is a strategy in which multiple layers of security are implemented to protect the company assets. The idea is that if one layer of security becomes vulnerable, the other layers should still be there to protect against the attackers. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques ensure that the products/services meet the highest standards and meet all stakeholder expectations. Some examples include automated tests, shift left, and test with purpose. |
| 1. Adopt a Secure Coding Standard | Adopting a set of secure coding standards ensures that code will be written to minimize any security vulnerabilities as much as possible. |

### 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 create incompatible declarations of the same function or object. |

| **Noncompliant Code** |
| --- |
| The code is noncompliant as it contains incompatible declarations of the variable a. In file D the variable a is declared as an int while in file E the variable a is declared as type long. |
| /\* File D \*/  extern int a;  int b(void) {  return ++a;  }  /\* File E \*/  long a; |

| **Compliant Code** |
| --- |
| The code is compliant as it now contains compatible declarations of the same variable. |
| /\* File D \*/  extern int a;  int b(void) {  return ++a;  }  /\* File E \*/  int a; |

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

| **Principle 10:** Adopt a Secure Coding Standard is linked to this Data Type coding standard because secure coding is essential to minimizing the vulnerabilities that can occur from incompatible declarations such as unintended information exposure, memory overwrite, or a hardware trap. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT C: Rule DCL40-C | Checks for declaration mismatch. |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.IF  LANG.STRUCT.DECL.IO | Checks for inconsistent function/object declarations. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensure that operations on signed integers do not result in overflow. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example displays how overflow can result from the addition of the signed integers a and b. |
| void addition(signed int a, signed int b) {  signed int sum = a + b;  } |

| **Compliant Code** |
| --- |
| This compliant code displays a solution to prevent overflow when adding two integers. |
| #include <limits.h>  void addition(signed int a, signed int b) {  signed int sum;  if (((b > 0) && (a > (INT\_MAX - b))) ||  ((b < 0) && (a < (INT\_MIN - b)))) {  /\* Handle error \*/  } else {  sum = a + b;  }  } |

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

| **Principles 1, 10:** Validate Input Data is applied because the possibly input integers are checked to ensure an overflow does not occur. Adopt a Secure Coding Standard applies because this method of checking and preventing certain integers from being added mitigates the security vulnerability of overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW | Addition/Integer/Multiplication overflow of allocation size Subtraction underflow of allocation size Addition/Multiplication overflow of size Unreasonable size argument Subtraction underflow of size |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| This noncompliant example showcases the char pointer str being initialized to the address of the string literal. Trying to change the string literal results in undefined behavior. |
| char \*str = "string literal example";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| This compliant code example shows the string literal being initialized in an array which specifies the values and size of the array. This allows the string literal to be modified. |
| char str[] = "string literal example";  str[0] = 'S'; |

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

| **Principle 10:** Adopt a Secure Coding Standard applies to this coding standard mitigates any vulnerabilities that could arise from modifying string literals such as the access violation or undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | String-literal-modification-write-to-string-literal | Checks for violations of this standard. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not log unsanitized user input. |

| **Noncompliant Code** |
| --- |
| This noncompliant code logs data from unauthenticated users without data sanitization. This could result in the forging of log entries or leaking secure information. |
| if (loginSuccessful) {  logger.severe("User login succeeded for: " + username);  } else {  logger.severe("User login failed for: " + username);  } |

| **Compliant Code** |
| --- |
| This compliant code sanitizes user data before to prevent the SQL injection of code being used to log into the system by using a method to search for certain characters within the usernames. |
| if (loginSuccessful) {  logger.severe("User login succeeded for: " + sanitizeUser(username));  } else {  logger.severe("User login failed for: " + sanitizeUser(username));  }  public String sanitizeUser(String username) {  return Pattern.matches("[A-Za-z0-9\_]+", username))  ? username : "unauthorized user";  } |

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

| **Principles 1, 7:** Validate Input Data applies to this standard because the log data is inspected and filtered out before it enters the log. Sanitize Data Sent to Other Systems applies to this because the user input is sanitized before it is logged. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2024.4 | SVLOG\_FORGING | Checks for log injection attacks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not write to or read from memory once it is freed. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example, s is dereferenced after being deallocated. This can result in a write-after-free and the vulnerability can be exploited to run arbitrary code with permissions of the vulnerable process. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the allocated memory is not deallocated until it is not needed anymore. |
| #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.**

| **Principle 10:** Adopt a Secure Coding Standard applies to Memory Protection because the code is written to minimize security vulnerabilities that arise from writing to or reading from memory that has been freed such as DoS attacks or arbitrary code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | V7.5.0 | USE\_AFTER\_FREE | This tool is able to detect instances where memory is deallocated more than once or read/written to a freed pointer. |
| Polyspace Bug Finder | R2024a | CERT C++: MEM50-CPP | Checks pointer access out of bounds, deallocation of previously deallocated pointer, and the use of a previously freed pointer. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Do not use assertions to verify the absence of runtime errors. |

| **Noncompliant Code** |
| --- |
| This noncompliant code tries using an assertion to verify that the line of input is not null. |
| String line;    line = br.readLine();    assert line != null; |

| **Compliant Code** |
| --- |
| This compliant code displays the correct way to check for and handle the unavailable line of input. |
| String line;    line = br.readLine();    if (line == null) {  // Handle error  } |

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

| **Principles 4, 10:** Keep it Simple applies because we are keeping the code simple and efficient by using an if statement to check that the line is null. Adopt a Secure Standard applies because we must use best practices rather than a failed assertion to verify if input was available because it can lead to vulnerabilities like denial of service. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Paarasoft Jtest | 2024.1 | CERT.MSC60.ASSERT | Checks for misuse of assertions. |

#### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the functions a() and main() do not catch the exception thrown by exception\_example(). Since no matching handler is found, the function std::terminate() is called. |
| void exception\_example() noexcept(false);    void a() {  exception\_example();  }    int main() {  a();  } |

| **Compliant Code** |
| --- |
| For this compliant code example, the main function handles the exception thrown by the exception\_example() function in a(). |
| void exception\_example() noexcept(false);    void a() {  exception\_example();  }    int main() {  try {  a();  } catch (...) {  // Handle error  }  } |

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

| **Principles 3, 9:** Architect and Design for Security Policies applies to this standard because the code must contain a proper try-catch block to test and catch the errors. Use Effective Quality Assurance Techniques applies to this standard because we are testing for exceptions to be found and handling those exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR51-a CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Write constructor member initializers in canonical order. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, A() tries to initialize value2 first and then initialize value3 to some value that's dependent on value2. The declaration order of the variables does not match the initializer order so an unspecified value is stored in variable value3. |
| class A {  int value3;  int value2;    public:  A(int value1) : value2(value1), value3(value2 + 1) {}  }; |

| **Compliant Code** |
| --- |
| This compliant code example fixes the declaration order of the member variables to match that of the member initializer order. |
| class A {  int value2;  int value3;    public:  A(int value1) : value2(value1), value3(value2 + 1) {}  }; |

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

| **Principle 4:** Keep It Simple applies to this coding standard because by writing the code in canonical order makes the code easier to understand and straightforward. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-OOP53-a | List members in an initialization list in the order in which they are declared |
| Polyspace Bug Finder | R2024a | CERT C++: OOP53-CPP | Checks for members not initialized in canonical order |

#### Coding Standard 9

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

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the file is open() but never matched with a call to close() the file before terminate() is called. This leads to the object not being 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** |
| --- |
| This compliant code example utilizes file.close() before calling terminate() so that the files resources are 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;  }  // ...  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 10:** Adopt a Secure Coding Standard applies to this standard because it is proper form to close any files that were opened to reduce the risk of exhausting available resources. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2024a | CERT C++: FIO51-CPP | Checks for resource leak |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer Overflow | [STD-010-CPP] | Prevents more than the allotted number of characters from being read by the console. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example displays an issue where the user can input more than 10 characters leading to a buffer overflow error. |
| const std::string account = "JohnDoe8";  char user\_input[10];  std::cout << "Enter a value: ";  std::cin >> user\_input;    std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account = " << account << std::endl; |

| **Compliant Code** |
| --- |
| This compliant code example displays the use of cin.getline to limit the number of characters read from the console to prevent a buffer overflow. |
| const std::string account = "JohnDoe8";  char user\_input[10];  std::cout << "Enter a value: ";  std::cin.getline(user\_input, 10);    std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account = " << account << std::endl; |

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

| **Principles 1, 8:** Validate Input Data applies to this buffer overflow situation because we must ensure that no more than 10 characters are read from the system by limiting the number of characters read to 10. Practice Defense in Depth is applied here because this is one layer that is used to prevent DoS attacks from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT++: CTR52-CPP | Checks for library functions overflowing sequence container. |
| CodeSonar | 8.1p0 | BADFUNC.BO.\*  LANG.MEM.BO  LANG.MEM.TBA | Runtime error detection of buffer overflow. |

### 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 DevSecOps image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Security automation should be considered a priority and should be enforced early and often in the software development lifecycle. Automation tools should be enforced because they also evade human errors. We can implement automation tools for continuous testing to occur within the CI/CD pipelines. This can begin in the build stage of the DevSecOps diagram. Security automation tools will include static application security testing (SAST) like Cppcheck to analyze the source code to find vulnerabilities or unsafe implementations of code. This will allow the team to enforce the coding standards by finding these issues as they arise and fixing them. In the verify and test stage, dynamic application security testing tools can be used to find vulnerabilities through simulated attacks. In the transition and health check stage, runtime verification tools can be used to determine whether the system runs as expected. Dealing with security in automation early is more effective than having to deal with potential issues later.

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

### Policies for Encryption and Triple A

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest works on data at rest, data that isn’t actively traveling, which could be databases or file servers. This data is important and is very valuable to hackers. The data at rest should be encrypted with complex algorithms such as AES-256. This will make it so that even if attackers obtain the data, it will be nearly impossible to decipher without the decryption key. |
| Encryption in flight | Encryption in flight refers to data traveling from one place to another. This data is at risk of being intercepted and stolen. Encryption in flight involves sensitive data being encrypted as it travels to its destination and being decrypted by the authorized receiver. This allows the data to be inaccessible when intercepted without the decryption key. |
| Encryption in use | Encryption in use refers to data being accessed or processed by users or software. Data becomes vulnerable at this stage. Encryption in use addresses this vulnerability by encrypting and decrypting data in real-time. This makes it so that data is encrypted during its entire lifecycle along with at rest and in flight so that no attackers can obtain plaintext data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying who someone is or is claiming to be. A user may identify themselves by using proper login credentials such as username, email, and password. New users will be added to the database along with their login credentials. This policy applies because whenever any user logs into the system they must be authenticated and checked with credentials in the database to defend against certain threats. |
| Authorization | Authorization is the process of determining what services a user is allowed to access. Once the user is authenticated, they are authorized with certain privileges depending on their status. This, along with the principle of least privilege, should keep the system protected by keeping users out of areas they should not be allowed to access. |
| Accounting | Accounting is the process of keeping track of what resources were accessed by who and at what time. Any files accessed by users or changes to the database can be tracked by session time, date, and which user accessed this data. This policy is beneficial when administrators perform audits to check on any unwanted actions. Tracking changes made to the system is important especially when any issues arise. |

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

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

## 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 | 11/21/2024 | Defined 10 security principles. Created 10 coding standards. | Fernando Lomeli | Fernando Lomeli |
| 1.2 | 12/12/2024 | Updated coding standards, added risk assessment, automation tools, and mapped principles. Added explanation of automation, summarized risk assessment table, and created policies for Encryption/Triple A. | Fernando Lomeli | Fernando Lomeli |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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