**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 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 means checking everything that a user enters in an input field to make sure it is safe and not malicious or code. This helps prevent hackers from sending harmful code or data that could break the program or steal information from the program. It is a very important and often overlooked principle of security but validating that input is safe is something that should be easy to implement. |
| 1. Heed Compiler Warnings | When the compiler shows a warning, usually that means it is trying to show that something is wrong in the code. Even if the program runs, checking the compiler warnings and fixing them early should be a priority. This prevents issues down the line, so that these warnings don’t turn into a security flaw later. |
| 1. Architect and Design for Security Policies | Security should be something thought of through every stage of the process. It should be considered at the start of a project, not just added towards the end. By creating a architect and design for security policies, security can be thought out and planned ahead control of access can be done right at the start instead of rushed and not done as well later. |
| 1. Keep It Simple | Keeping security policies and procedures simple is a good way to make it easy for any user or developer to understand. The more complicated security gets, the harder it is to maintain. It also leads to more chances of things breaking or going wrong, especially if security is overlooked. |
| 1. Default Deny | The default deny rule means that access isn’t specifically allowed then it will be denied. This is a way of avoiding someone gaining access that other wise should not have gotten access to the system. Anyone that is not given explicit permission won’t have permission and the system will be safer for it. |
| 1. Adhere to the Principle of Least Privilege | This means only giving users access to programs they need to do their jobs. One way of implementing this is Role Based Access Control. This way if something does get hacked the damage will be smaller, because there is less access for the user that got hacked. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to another system, we should clean up the data to make sure it doesn’t contain anything malicious. This just helps avoid passing along harmful code to another system. |
| 1. Practice Defense in Depth | Defense in depth is about using multiple layers of security so that if one layer of security fails there are other layers in place to protect the system. I view this as home security where I have a door lock, a security camera, an alarm, and flood lights. All four things are a layer of security I’ve added to my house. The same can be done in cybersecurity, where there are many layers of security that act independently so if one fails, there are others to take the spot. |
| 1. Use Effective Quality Assurance Techniques | Good code testing is a great way to catch a problem before it becomes a problem. Another option is having good code reviews and regularly reviewing the code so that safer code can be written. Another option is to include vulnerability checks or dependencies checks to ensure that libraries being used are up to date, as using out of date libraries can lead to security issues. |
| 1. Adopt a Secure Coding Standard | Using a secure coding standard means following the standards with cybersecurity to avoid common mistakes. These standards are sort of like best practices, while they are not enforced (unless you work for the DoD) a good starting point is the NIST 800-53 or NIST 800-171 controls. These controls are set up to ensure good cyber security practices and help industries think of security in all aspects, all the time. |

### 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] | Name: Validate Input Data Types  Rationale: Validating input data types prevent security vulnerabilities like injection attacks, and also can prevent type issues within the code. Ensuring that the data type expected is received helps maintain the system and security of it. |

| **Noncompliant Code** |
| --- |
| This accepts user input without checking if it is a valid input for example if the user inputs a string, the program will throw an error. |
| int main(){  int age;  cout << “Enter age”:  cin >> age;  of (age>18){  cout<< “You are of age”>> endl;  }  Return 0;  } |

| **Compliant Code** |
| --- |
| This validates the input to make sure it is an int, and if the user inters invalid data types, the program will be able to handle it. |
| int main() {  int age;  cout << "Enter your age: ";  cin.clear();  if (!(cin >> age)) {  cout << "Invalid input. Please enter a number." << endl;  return 1;  }  if (age > 18) {  cout << "You are an adult." << endl;  }  return 0;  } |

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

| **Principles(s):** Validate Input Data Types – This principal maps to this standard which confirms data types are valid to prevent data type mismatches. This makes sure that only expected data types is being used by the program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.17 | Type mismatch, Input validation | Static analysis tool detecting unsafe input handling |
| SonarQube | |  |  | | --- | --- | |  | 4.19 | | |  | | --- | |  |  |  | | --- | | cpp:S5838 | | Flags lack of input validation |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Name: Avoid Magic Numbers  Rationale: Using unnamed numeric constants in code can make is harder to update or debug later. Replacing these numbers with named constants make the code easier to understand which makes security of the code easier and reduces the chance of unintended logic errors which enhances security. |

| **Noncompliant Code** |
| --- |
| This uses hardcoded value without any reason to why, the number 60 is unclear on what it is or why it is being used which means if this code needs to be updated it may be confusing. |
| int main() {  int time = 75;  if (time > 60) {  cout << "Time limit exceeded!" << endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This uses a named constant instead which makes it easier to maintain and makes it more readable. |
| const int TIME\_LIMIT = 60;  int main() {  int time = 75;  if (time > TIME\_LIMIT) {  cout << "Time limit exceeded!" << endl;  }  return 0;  } |

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

| **Principles(s):** Keep it simple – This security control maps to the principle keep it simple because by replacing un named constants with named constants the code becomes easier to read and understand. Simpler code reduces bugs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 4.19 | cpp:S109 | Identifies use of magic numbers in code |
| Cppcheck | 2.17 | style:magicNumber | Detects magic numbers and suggests using constants |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Name: Ensure Strings Have Enough Space for Null Terminator  Rationale: In C++ strings represented as character arrays must have enough space for both the string data and the null terminator so we must allocate space for it to avoid issues. Developers should use std::string, which manages memory automatically and will avoid direct memory handling risks. |

| **Noncompliant Code** |
| --- |
| This does not leave room for the null terminator. |
| int main() {  char name[4];  strcpy(name, "John");  return 0;  } |

| **Compliant Code** |
| --- |
| This does leave room for the null terminator. |
| int main() {  char name[5];  strcpy(name, "John");  return 0; |

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

| **Principles(s):** Defense in Depth - This security standard maps to the principal practice defense in depth by the use of std:string to handle memory management safely. It also maps to validate input data because proper memory allocation for strings makes sure that inputs are stored safely. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | bufferOverrun | Detects buffer overflows due to improper array allocation |
| SonarQube | 4.19 | cpp:S3518 | Flags use of unsafe string manipulation and missing null-terminator |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Name: Use Prepared Statements to Prevent SQL Injection  Rationale: SQL injection is a security risk that allows attackers to inject malicious code through user input, by using prepared statements we can ensure that executable code will not make it into input fields. This is not specific to SQLite but other database platforms as well like MySQL. |

| **Noncompliant Code** |
| --- |
| This builds the SQL query by inserting user input into the string which makes it vulnerable to an SQL injection. |
| void getUserData(sqlite3\* db, const std::string& username) {  std::string query = "SELECT \* FROM users WHERE username = '" + username + "';";  sqlite3\_exec(db, query.c\_str(), nullptr, nullptr, nullptr);  } |

| **Compliant Code** |
| --- |
| This uses a prepared statement which separates the code form data and protects against SQL injection. |
| void getUserData(sqlite3\* db, const std::string& username) {  sqlite3\_stmt\* stmt;  const char\* sql = "SELECT \* FROM users WHERE username = ?;";    sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_STATIC);  sqlite3\_step(stmt);  sqlite3\_finalize(stmt);  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems: This security standard maps to sanitize data sent to other systems since prepared statements sanitize SQL input before it gets to the database. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 4.19 | cpp:S2077 | Flags unsanitized user input in SQL queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Name: Do Not Access Memory Outside of Object Lifetime  Rationale: Accessing memory after an object has been deleted can cause crashes or security vulnerabilities. To ensure that all memory accessed is within valid object lifetimes, this standard must be followed. It is recommended to use smart pointers like std::unique\_ptr to manage memory allocation automatically rather to reduce the risks associated with manual allocation. |

| **Noncompliant Code** |
| --- |
| This deletes a dynamically allocated object and then attempts to access the data which will cause issues. |
| int\* createData() {  int\* data = new int(10);  delete data;  return data;  }  int main() {  int\* value = createData();  cout << \*value << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This avoids accessing the memory after the object has been deleted and only access memory if it is a valid object. |
| int\* createData() {  int\* data = new int(10);  return data;  }  int main() {  int\* value = createData();  cout << \*value << endl;  delete value;  return 0;  } |

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

| **Principles(s):** Practice Defense in Depth – This security standard maps to practice defense in depth by using smart pointers to make sure that memory is only accessed when valid. It also maps to Use Effective Quality Assurance Techniques as this standard supports quality assurance by forcing safe memory practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | useAfterFree | Detects use of freed pointers |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Name: Avoid Using assert() for Runtime Error Checking  Rationale: Assertions are meant to catch programming logic errors during development. Assert() is unreliable and unsafe for detecting runtime errors because asset() can be disabled in production. |

| **Noncompliant Code** |
| --- |
| This uses assert() to check for a runtime condition. |
| void printMessage(const char\* message) {  assert(message != nullptr);  std::cout << message << std::endl;  } |

| **Compliant Code** |
| --- |
| This uses a proper runtime check instead of assert(). |
| void printMessage(const char\* message) {  if (message == nullptr) {  std::cerr << "Error: message is null." << std::endl;  return;  }  std::cout << message << std::endl;  } |

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

| **Principles(s):** Heed Compiler Warnings – This security standard maps to heed compiler warnings by replacing assert() with proper runtime checks and prevents assert() from being disabled in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | assertUsage | |  | | --- | |  |  |  | | --- | | Warns when assert() is used in  place of runtime validation | |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Name: Use Exceptions for Error Handling  Rationale: Using exceptions for error handling allows the program to respond to errors easier and maintain more control. |

| **Noncompliant Code** |
| --- |
| This uses the exit() to handle an error which halts the program |
| void openFile(bool success) {  if (!success) {  std::cerr << "Failed to open file." << std::endl;  exit(1);  }  } |

| **Compliant Code** |
| --- |
| This uses a try/catch block with exceptions which allows the program to better handle the error. |
| void openFile(bool success) {  if (!success) {  throw std::runtime\_error("Failed to open file.");  }  }  int main() {  try {  openFile(false);  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Keep it simple – This security standard maps to keep it simple because using exceptions allows for error handling which makes the program cleaner. It also maps to use effective quality assurance techniques because exceptions are a good way to catch and handle errors in development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | useStlException | Warns about missing exception handling or unsafe termination |
| SonarQube | 4.17 | cpp:S3516 | Flags improper or missing use of exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Initialization | [STD-008-CPP] | Name: Declare Static Objects in Block Scope  Rationale: Objects with static storage should be declared inside a function because it limits their visibility and reduces the risk of unintended side effects. |

| **Noncompliant Code** |
| --- |
| This code declares a static variable at the global level which increases risk |
| static int counter = 0;  void increment() {  counter++;  std::cout << counter << std::endl;  } |

| **Compliant Code** |
| --- |
| This has the static variable declared inside the function which limits the scope of where it is needed and used. |
| void increment() {  static int counter = 0; // Block scope  counter++;  std::cout << counter << std::endl;  } |

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

| **Principles(s):** Keep it simple – This security standard maps to keep it simple because by limiting the scope of static variables within the block they are needed makes the code easier to understand and maintain. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | variableScope | Detects overly wide variable scopes |
| SonarQube | 4.19 | cpp:S1820 | Flags variables declared with wider scope than necessary |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | [STD-009-CPP] | Name: Do Not Read from Uninitialized Pointers  Rationale: Using uninitialized pointers can cause unexpected behavior. Pointers should be initialized before use. |

| **Noncompliant Code** |
| --- |
| This code attempts to use a pointer that has not been initialized. |
| int main() {  int\* ptr;  std::cout << \*ptr << std::endl;  } |

| **Compliant Code** |
| --- |
| This code makes sure the pointer is initialized |
| int main() {  int value = 42;  int\* ptr = &value;  std::cout << \*ptr << std::endl;  return 0;  } |

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

| **Principles(s):** Defence in depth – this security standard maps to defense in depth because making sure that pointers are initialized before use adds a safeguard to prevent memory access violations. This will create more secure code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | uninitvar, nullPointer | Detects uninitialized variables and unsafe pointer usage |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Namespace Usage | [STD-010-CPP] | Name: Avoid using namespace std; in Header Files  Rationale: Using namespace std in header files will bring all standard library names into the global scope which may cause naming issues. |

| **Noncompliant Code** |
| --- |
| This header file is using namespace std; |
| using namespace std;  string greet(const string& name); |

| **Compliant Code** |
| --- |
| This keeps the std:: prefix, which avoids polluting the global namespace. |
| std::string greet(const std::string& name); |

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

| **Principles(s):** Keep it simple – This security standard maps to keep it simple because by not using namespace std in headers the code is clearer and won’t introduce unintended names to the global scope which makes it easier to maintain. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17 | usingNamespace | Warns when using namespace is declared in header files |

### 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.

Green Pace can improve its compliance with the standards in this policy by the following:

* Using automation tools like CppChecker to identify any areas of weakness and improve code security practices.
* Use secure standards like Cert C++.
* Use trusted libraries and maintain updated versions.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Medium | Low | High | 4 |
| STD-002-CPP | Medium | High | Low | Medium | 4 |
| STD-003-CPP | High | High | Low | High | 5 |
| STD-004-CPP | High | High | Medium | High | 5 |
| STD-005-CPP | High | Medium | Medium | High | 3 |
| STD-006-CPP | High | Medium | Low | Medium | 2 |
| STD-007-CPP | Medium | Medium | Low | Medium | 3 |
| STD-008-CPP | Medium | Medium | Low | Medium | 2 |
| STD-009-CPP | High | Medium | Low | High | 4 |
| STD-010-CPP | Medium | High | Low | Medium | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects data that is stored on a physical device such as a hard drive. Sensitive information like passwords are stored using a strong encryption algorithm like AES-256. This is to prevent unauthorized access to the data at rest. |
| Encryption in flight | Encryption in flight refers to securing data as it is being transmitted over the network. TLS/SSL should be used for encryption in flight to prevent interception. |
| Encryption in use | Encryption in use is when data is being processed in the memory. This should be done in secure enclaves to prevent unauthorized access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the verification of the identity of the person trying to access the data or system. There should be Role Based Access Controls in place to prevent unauthorized access going off the principle of least privilege. This will help to prevent compromised accounts. |
| Authorization | Authorization refers to what actions authenticated users are allowed to perform. A regular user may not have admin privileges, and any admin actions may need to be authorized. Role based access control also applies here and helps limit compromising an account. |
| Accounting | Accounting refers to tracking activities and reviewing logs. All user sessions should be logged for review. This helps with auditing and compliance. Any changes to a system such as a database would be log so changes can also be tracked. |

**\***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 | 4/11/2025 | Updated all sections | Ally Miller | Ally Miller |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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