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



# Green Pace Secure Development Policy

# Completed by Sarah Kerr

# CS 405

## 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 | The process of taking input from a source that cannot be trusted (whether its user supplied input or from another application) and checking that input against a standard before accepting it. This prevents issues like memory leaks and security vulnerabilities that could occur if unexpected input is passed into the program. |
| 1. Heed Compiler Warnings | Compiler warnings are one of the first steps in insuring secure code. They can detect possible security issues even if the code is technically sound. Compiler warnings should be dealt with as they occur rather than waiting until after the program has been written. This will prevent having to take unnecessary time later fixing several lines of code. |
| 1. Architect and Design for Security Policies | This refers to designing with security in mind. Having a security mindset from the beginning of a project will greatly improve the defense of that program. This includes things like giving appropriate user permissions and providing extra security for sensitive data. |
| 1. Keep It Simple | Code that is simple is easy to read and understand. Good defensive programming cannot be applied to a program that is not well understood by others, so it is important to avoid unnecessary complexity. Use naming conventions that are easy to understand and modulate the program as much as necessary to prevent code reuse. |
| 1. Default Deny | By default, a user shall not be given any permission to access any part of the software. Those permissions can only be granted through user authorization. This provides a layer of security to the software because the user must prove that they have authorization. |
| 1. Adhere to the Principle of Least Privilege | Users should be granted as much access as they need to perform their tasks and no more. There should be only one user with administrative privileges. This not only protects against security threats but also by a user mistakenly changing something. |
| 1. Sanitize Data Sent to Other Systems | All output data should be encoded to be properly used by subprograms and other programs. This means that if CSS is being used, the data needs to be encoded as CSS. |
| 1. Practice Defense in Depth | This means adding multiple layers of security to the software to ensure full coverage. No security policy by itself can provide full coverage, so several strategies are used together. This also means that redundancies will occur. |
| 1. Use Effective Quality Assurance Techniques | It is important to use the most effective security measures. As time goes on, more effective security measures are likely to be needed. Following security guides such as the OWASP SCP can assist in understanding the most up-to-date security standards, and company guidelines should be updated as necessary. |
| 1. Adopt a Secure Coding Standard | Using a standard such as this as a guide will ensure that all team members are following the same practices. This will reduce the amount of time needing to update code for security later on. This will also assist in code reviews because it offers a guideline to follow. |

### 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** | **Validate Data Type** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Input data must be checked for correct type or the system could crash or run into other issues. |

| **Noncompliant Code** |
| --- |
| The code here is vulnerable because there is no protection against the data type the user is entering. The program is expecting int but the user could easily enter a string. |
| int x;  cin >> x;  cout << x; |

| **Compliant Code** |
| --- |
| Now the input is validated by an if statement. !cin indicates a data type entered that was not expected. |
| int x;  if (!cin){  cout << “Incorrect data type”;  cin.clear()  cin.ignore(numeric\_limits<streamsize>::max(), ‘\n’)  cin >> x;  } else {  cout << x;  } |

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

| **Principles(s):**  Principle 1 - Validate Input Data: Validating that the correct data type has been entered is one way of validating input  Principle 8 – Practice Defense in Depth: Always apply security in places where input is coming in. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | LANG.CAST.COERCE  **LANG.CAST.VALUE** | Coercion alters value  Cast alters value |
| CppCheck | 1.66 | **memsetValueOutOfRange** | The second argument to memset() cannot be represented as unsigned char |

#### 

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate Data Values** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Data values must be checked against their expected ranges or an overflow or underflow could occur. |

| **Noncompliant Code** |
| --- |
| The below code will result in a wraparound because the upper range limit of int is 2147483647. X will display as a negative number. |
| int x = 2147483647;  x += 1;  cout << x; |

| **Compliant Code** |
| --- |
| std::numeric\_limits is used to the get the maximum number for int. This code will check the limit against the value to be added (in this case, the variable ‘increment’) before actually adding the two numbers. This catches the overflow and also prevents the program from going into Undefined Behavior. |
| int x = 2147483647;  int increment = 1;  if (x > 0 && std::numeric\_limits<int>::max() – increment){  cout >> “Error: Overflow detected”;  }  else {  result = x + increment;  } |

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

| **Principles(s):**  Principle 1 – Validate Input Data: Validate by checking that performing an operation on a value won’t cause an overflow *before* performing operation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | **ALLOC.SIZE.IOFLOW** | Integer overflow of allocation size |
| Klocwork | 2022.2 | [NUM.OVERFLOW](https://docs.roguewave.com/en/klocwork/2021/num.overflow) | detects overflow events when a variable is incorrectly typecast. |
| Coverity | 2017.07 | **INTEGER\_OVERFLOW** |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Check the size of a string before copying into array** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | This prevents a user attempting to enter a string into a C array that is not large enough to hold it. Doing so would cause a buffer overflow. |

| **Noncompliant Code** |
| --- |
| The code attempts to copy a string to an array that will not hold it, resulting in a buffer overflow. |
| char myArray[20];  std::string myString = “This will have more than 20 char”;  strcpy(myArray, myString); |

| **Compliant Code** |
| --- |
| The input is first taken as a string. Strlen() is used to get the size of the string and sizeof() is used to get the size of the array. These two are first compared to make sure the input can fit in the array. If it can, strcpy() is used to copy the string to the array. |
| char myArray[20];  std::string myString = “This will have more than 20 char”;  if (strlen(myString) >= sizeof(myArray) {  cout << “Error: Buffer overflow”;  }  else {  strcpy(myArray, myString);  } |

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

| **Principles(s):**  Principle 1 – Architect and Design for Security: Design code that checks for final string size before attempting to copy into array. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | **LANG.MEM.BO** | Detects buffer overrun |
| Coverity | 2017.07 | **STRING\_OVERFLOW** | Detects buffer overflow of string |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection from the user** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | A malicious user can gain access of the database by injecting statements that would circumvent authorization protocols. This standard protects against that. |

| **Noncompliant Code** |
| --- |
| This code shows a malicious user entering a query with ‘or 1=1’ at the end which will cause the statement to return regardless of whether ‘wanteddata’ exists or not. This could give a malicious user access to the database. |
| std::string sqlQuery = “SELECT \* FROM database WHERE somedata=’wanteddata' or 1=1”;  for (auto sqldata : database) {  std::cout << std::get<0>(sqldata);  } |

| **Compliant Code** |
| --- |
| Before the query can be executed, the sqlQuery string is checked for any instances of “ or “ which would indicate a user trying to pass extra parameters to gain access to the database. |
| std::string sqlQuery = “SELECT \* FROM database WHERE somedata=’wanteddata' or 1=1”;  if (sqlQuery.find(“ or “) != std::string::npos) {  std::cout << “Error, possible SQL injection detected”;  }  else {  for (auto sqldata : database) {  std::cout << std::get<0>(sqldata);  }  } |

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

| **Principles(s):**  Principle 1 – Validate Input Data: Check SQL queries for indicators of possible malicious injection.  Principle 6 – Adhere to Principle of Least Privilege: Do not give users the right to directly enter commands into database. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | High | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarCube | 9.5 | [Database queries should not be vulnerable to injection attacks](https://rules.sonarsource.com/python/RSPEC-3649) | User-provided data, such as URL parameters, should always be considered untrusted and tainted |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Set Pointers to Null After Deferencing** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | All dereferenced pointers should be set to NULL to protect against writing to freed memory and double free vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The below code creates a pointer to value and then dereferences it when it is output. Failing to set the pointer to NULL afterward leaves the program vulnerable. |
| int value = 5;  int\* ptr = &value; //Pointer to value    std::cout << \*ptr; //Dereference of pointer without ensuring null after  free (ptr); |

| **Compliant Code** |
| --- |
| The pointer is now set to NULL after it is freed |
| int value = 5;  int\* ptr = &value; //Pointer to value    std::cout << \*ptr;  free (ptr);  ptr = NULL; |

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

| **Principles(s):**  Principle 7 – Sanitize Data Sent to Other Systems: Always make sure to set dereferenced pointers to NULL to prevent misuse by other areas of the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | **ALLOC.UAF** | Use after free |
| Coverity | v7.5.0 | **USE\_AFTER\_FREE** | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Check Input** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to verify expected input when in debug mode. |

| **Noncompliant Code** |
| --- |
| The following code is expecting a positive number but there is no check to make sure this is true. |
| int x = FunctionReturningPosNum(y); |

| **Compliant Code** |
| --- |
| The use of ASSERT ensures that the function is returning expected values. If not, an error is displayed. |
| int x = FunctionReturningPosNum(y);  ASSERT(x >= 0); |

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

| **Principles(s):**  Principle 2 – Heed Compiler Warnings: Assertions take place in the compiler in debug mode, so developers will need to heed any issues they detect  Principle 9 – Use Affective Quality Assurance: The use of assertions during development and testing is one way to perform QA by checking for expected input.  Principle 10 – Adopt a Secure Coding Standard: Since the use of assertions is not required for product deployment, this is strictly a secure coding practice during development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Microsoft Visual Studio | 2022 | CRT assertions | One of many examples of compilers that have assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Use Noexcept for Exception Handling** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | throw() was previously used, but this was found to have vulnerabilities and can cause Denial of Service attacks. It is recommended to use noexcept(). |

| **Noncompliant Code** |
| --- |
| This code tells the compiler there are no exceptions. However, if there is an exception, this will cause Undefined Behavior. |
| void aFunction(int x) throw(); |

| **Compliant Code** |
| --- |
| noexcept() is the same as noexcept(true) so if there is an exception, the program will terminate instead of going into UB. |
| void aFunction(int x) noexcept(); |

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

| **Principles(s):**  Principle 4 – Keep It Simple: noexcept() is simpler to use and has less issues than throw() |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2022.1 | **CERT\_CPP-ERR60-a** | Exception objects must be nothrow copy constructible |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not leave pointers uninitialized** |
| --- | --- | --- |
| **Pointers** | [STD-008-CPP] | Referencing to uninitialized pointers can cause a lot of vulnerabilities because it accesses random memory addresses. Always initialize a pointer when creating it. |

| **Noncompliant Code** |
| --- |
| The pointer is not initialized before it is called on. |
| int\* ptr;  std::cout << &ptr; |

| **Compliant Code** |
| --- |
| Now the code will be checked to see if it’s null before continuing. |
| int\* ptr = NULL;  if (\*prt != NULL) {  std::cout << &ptr;  } |

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

| **Principles(s):**  Principle 2 – Heed Compiler Warnings: Most compilers will flag an uninitialized variable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | v7.5.0 | **USE\_AFTER\_FREE** | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Enforce Password Lengths** |
| --- | --- | --- |
| **Authentication** | [STD-009-CPP] | Ensure the user has to enter a minimum password length. Most applications require at least 8 characters. |

| **Noncompliant Code** |
| --- |
| The code does not enforce a password length so the user could type one character which would not be very secure. Note, using getline() ensures no more than 30 characters are read. |
| char userPassword[30];  std::cin.getline(userPassword, 30); |

| **Compliant Code** |
| --- |
| The code checks if the password length is at least 8 characters in length. |
| char userPassword[30];  std::cin.getline(userPassword, 30);  if (userPassword.length() < 9) {  std::cout << “Error, password must be at least 8 characters in length”;  std::cin.ignore();  }  else {  // Continue  } |

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

| **Principles(s):**  Principle 8 – Practice Defense in Depth: Requiring strong passwords can add one layer of security  Principle 10 – Adopt a Secure Coding Standard: This practice will help increase security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Not applicable for this standard |  | Not applicable for this standard |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not allow user to enter conversion specifics in formatted output** |
| --- | --- | --- |
| **Formatted Output** | [STD-010-CLG] | Formatted output needs to be checked that the user didn’t try to enter a conversion specifier. |

| **Noncompliant Code** |
| --- |
| printf() is used but there is nothing to prevent the user from entering a conversion specifier like ‘%p’ which could cause information from memory to be written |
| char output[100];  char getOutput() {  /\* Range checking should happen here first \*/    printf(pass);  } |

| **Compliant Code** |
| --- |
| The insertion of ‘%s’ into the code prevents the user from entering conversion specifiers. This means the string will be read directly. |
| char output[100];  char getOutput() {  /\* Range checking should happen here first \*/    printf(‘%s’, pass);  } |

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

| **Principles(s):**  Principle 1 – Validate Input Data: Ensuring user cannot enter string formatters is one way to validate input  Principle 5 – Default Deny: Although some users may use string modifiers for innocent purposes, it is best to assume that all string modifiers are for malicious purposes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| Parasoft C/C++ test | 2022.1 | **CERT\_C-STR30-a**  **CERT\_C-STR30-b** | A string literal shall not be modified  Do not modify string literals |

### 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.

**The goal of DevSecOps is to address security issues right from the beginning. Therefore, more emphasis on security will occur in the pre-production phase. The creation of secure coding standards as outlined in this document is part of the assessment and plan phase of pre-production. It is important to do threat modeling of the project during this phase as well. When designing and building, secure practices must be kept in mind and using tools such as static analyzers is implemented from the beginning. More formal unit tests can then take place during the verification and testing phase before production. From there, feedback is gleaned and used in future planning phases.**

### 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 | Probable | High | Medium | 2 |
| [STD-002-CPP] | High | Likely | High | Medium | 2 |
| [STD-003-CPP] | High | Likely | Medium | High | 1 |
| [STD-004-CPP] | High | High | High | High | 1 |
| [STD-005-CPP] | High | Likely | Medium | High | 1 |
| [STD-006-CPP] | Low | Unlikely | Low | Low | 3 |
| [STD-007-CPP] | Low | Probable | Medium | Low | 3 |
| [STD-008-CPP] | High | Likely | Medium | High | 1 |
| [STD-009-CPP] | High | High | Low | Medium | 2 |
| [STD-101-CLG] | Low | Likely | Low | Medium | 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 | Keeping data encrypted when it is stored (such as on a disk) so that a hacker still cannot gain access to the information. |
| Encryption at flight | Encrypting data when it is being transferred across a network prevents third parties from intercepting the data. |
| Encryption in use | Encrypted data even when it is being used in a database such as with the use of hashing |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The user proving who they say they are during user login, usually with a username and password combination. This ensures only users you want accessing the system can access it. |
| Authorization | Once a user gains access, they are only allowed to see parts of the system that are applicable to them. This keeps data private by limiting the users who have access it to only those who need to access it. |
| Accounting | The process of logging system access, time of access, and what occurred. Reviewing these reports can offer important information on potential breaches or gaps in security. |

**\***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 | 07/17/2022 | Created security principles and 10 coding standards | Sarah Kerr |  |
| 2.1 | 08/07/2022 | Security Policy Complete and Ready for Use | Sarah Kerr |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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