**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 | User input should never be trusted. Instead, it should be limited by using a white-list approach. Contrary to a black-list approach, the program should only accept input that is exclusively defined by the programmer (i.e. type, size, etc.). |
| 1. Heed Compiler Warnings | Don’t write-off compiler errors at build time. Using deprecated functions can lead to unnecessary security vulnerabilities. A program should always be optimized with the latest versions of functions in a library. As an example, Visual Studio will throw a warning when using the unsafe “scanf” function and prompt the programmer to use the more robust “scanf\_s”. |
| 1. Architect and Design | Adhering to the security principles, policies, guidelines, and standards set forth by this document is paramount to maintaining security. Creating a framework for developers to work within will allow company best standards to guide the development of software. A team of individuals should audit and monitor the production of software to ensure policies are being upheld. |
| 1. Keep It Simple | Over complication can obscure inherent flaws in code. With that, if the readability of the code is poor it may take additional time to troubleshoot the root cause of an exploit, if one was to happen. |
| 1. Default Deny | By default, users should not have access. They should only be permitted once they have been authenticated and authorized. By doing so, risk of unwarranted entry is mitigated. |
| 1. Adhere to the Principle of Least Privilege | Users should have clearly defined roles. These roles dictate what they are able to do, such as ‘read’, ‘write’, or both. A user should be given only as much privilege as needed. |
| 1. Sanitize Data Sent to Other Systems | Before interacting with databases or other systems, input from users should be formatted. Along with validating using a white-list approach, the input from a user should be converted into a standard format for the system it is being used with. For example, converting input from a user into a MongoDB query before being sent to the database. |
| 1. Practice Defense in Depth | Multiple layers of security should be used to promote a more robust security infrastructure. That way, if one layer fails, another layer may stop the attack. These layers can be divided into administrative controls, physical controls, and technical controls. |
| 1. Use Effective Quality Assurance Techniques | Using static tests and assertions on code should be common practice for all developers. The use of a dependency checker is also useful when ensuring that all dependencies are up to date, and do not possess any preventable exploits. Third-party testing of code should be done before ever being deployed into production. |
| 1. Adopt a Secure Coding Standard | By following a secure coding standard, a universal and seamless approach to development will occur among all employees. The secure code standard should be clear and easy to implement. Adherence to this standard will promote a safety-first development mindset. |

### 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** | **Never qualify a reference type with const or volatile.** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Trying to assign a new value to reference variable could result  in unexpected behavior from the program. |

| **Noncompliant Code** |
| --- |
| Trying to const-qualify a type as part of a declaration that uses a reference type. |
| #include <iostream>    void f(char c) {  const char &p = c;  p = 'p'; // Error: read-only variable is not assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| Removing the const qualifier allows for the variable to be assignable. |
| #include <iostream>    void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):** Heed Compiler Warnings.  In the case of this coding standard, a compiler warning will be thrown when trying to reassign a new value to a const variable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14.0 | variable can be declared const | Tool checks if the variable can be declared const, error will be thrown when const value is changed |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not read uninitialized memory** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Attempting to read variables before they are initialized can produce unexpected behavior due to the indeterminate value given to objects at declaration. |

| **Noncompliant Code** |
| --- |
| In this example, an uninitialized local variable is trying to be printed out to the user. This will result in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| Initializing the variable before the cout call will result in a error-free execution. |
| #include <iostream>    void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** Keep It Simple, Heed Compiler Warnings.  It may seem tedious, but initializing a variable at the time of declaration removes the possibility of forgetting to initialize it, causing undefined behavior if a variable is called at runtime. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14.0 | Bounds checking | Throws error when it finds an attempt to use a variable that has not been initialized, memory amount is indeterminate |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Providing insufficient space during string or character array declaration can result in buffer overflow. Buffer overflow is easily exploitable by attackers. |

| **Noncompliant Code** |
| --- |
| Calling cin on a char array of size 12 bytes can result in a buffer overflow if the user enters an input larger than 12 bytes. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The easiest way to avoid char array size limitations is by using the dynamic data type std::string. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Validate Input Data  Input gathered from a user should not be loosely defined. The program should only accept input that is within the parameters that it is looking for, all other input should be discarded. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.14.0 | Bounds Checking | Prevention of buffer overflow |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Detect errors when getting an invalid input from user** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Error handling should always be done immediately after an error occurs. If a user tries to enter an input of an invalid type, or contains restricted keywords, an exception should be thrown and handled immediately. |

| **Noncompliant Code** |
| --- |
| Input to cin is not limited, so if a string is entered undefined behavior may occur. |
| #include <iostream>    void f() {  int i, j;  std::cin >> i >> j;  // ...  } |

| **Compliant Code** |
| --- |
| Try-catch block set up to handle exceptions thrown if an invalid type is input. |
| #include <iostream>    void f() {  int i, j;    std::cin.exceptions(std::istream::failbit | std::istream::badbit);  try {  std::cin >> i >> j;  // ...  } catch (std::istream::failure &E) {  // Handle error  }  } |

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

| **Principles(s):** Validate Input Data, Sanitize Data Sent to Other Systems  Along with limiting what the program accepts as input from the user, the program should be able to efficiently handle a situation where a user does enter unaccepted input. Exceptions should be thrown and immediately caught and resolved, either by program termination or discarding the input. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P15 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.14.0 | Assert | Error is given when an invalid input is received for function using assert |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Accessing freed memory results in undefined behavior. Worse, pointers to memory that has been deallocated are known as dangling pointers and are susceptible to being exploited. |

| **Noncompliant Code** |
| --- |
| This example shows creating a new pointer of type S, deleting it, and then trying to access function f() after being deleted. This results in undefined behavior. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| To fix the example, just call the function with the pointer before deleting it. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** Heed Compiler Warnings, Use Effective Quality Assurance Techniques  Warnings should be thrown from the compiler when trying to use pointers that have been deallocated, but it is best practice to not leave any dangling pointers in production code. Use of static tests can identify these vulnerabilities as well. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14.0 | Memory Leaks | Warning is given when attempting to access a pointer or memory that has been previously deallocated. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Toggle assertions within code** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Remove assertions within production code, or at a minimum toggle them off.  Leaving assertions in can lead to exploits from injection attacks. |

| **Noncompliant Code** |
| --- |
| Assert has been left in on a production build. |
| void f()  {  std::vector<int> myVec\* = std::vector<int>(3);  assert(myVec != nullptr);  } |

| **Compliant Code** |
| --- |
| Assertion has been removed. Alternatively the assertions could be moved to a test case file. |
| void f()  {  std::vector<int> myVec\* = std::vector<int>(3);  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques, Architect and Design  Even though assertions are great for testing the functionality of code, they should not remain in production code. Best practice is to keep assertions out of all code that will be deployed to users. Implementing automated static tests into the design process will allow for a more efficient and secure SDLC. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.14.0 | Assertions | Assertions can be toggled off or removed from production build. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions should be handled at the time they occur. Exceptions that cannot restore the program to a stable state must terminate the program. |

| **Noncompliant Code** |
| --- |
| No exception handling on the function being used in Main, or within the f() function. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| Try – catch block inserted into main to handle exception if one is thrown from the f() function call. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** Adhere to the Principle of Least Privilege, Practice Defense in Depth  By adhering to the principle of least privilege, exploitation of invalid input can be mitigated even further when using exceptions. This follows with defense in depth, as having multiple layers of security is the best way to prevent unexpected behavior from a program. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14.0 | Exception Safety | Checks that all exceptions are handled. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| File Stream | [STD-008-CPP] | Corruption to files can happen if an input file stream is left open. Also, data may be leaked if an output file stream is left open. File streams must be properly closed and deconstructed. |

| **Noncompliant Code** |
| --- |
| In this example, std::terminate is used but this function does not call destructors. |
| #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** |
| --- |
| Using the close() function ensures that destructors are used on the file stream. |
| #include <exception>  #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):** Keep It Simple, Default Deny  As the principle states, it is most simple to close the file after all actions have been performed with the data inside of it. Keeping a file open can lead to corruption of the data within it. Also, permission to open/alter/close any file should be strictly prohibited unless user permissions are setup for such changes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14.0 | IO using format string | Checks for file stream handling |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not modify the standard namespaces** |
| --- | --- | --- |
| Namespaces | [STD-009-CPP] | Attempting to override the standard namespaces will result in undefined behavior. If wanting to create a new namespace, one must use a non-reserved name. |

| **Noncompliant Code** |
| --- |
| In this example, the reserved namespace “std” is being overloaded with a new definition, resulting in undefined behavior. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| To fix this, the namespace is changed to a non-reserved name. |
| namespace nonstd {  int x;  } |

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

| **Principles(s):** Keep It Simple, Heed Compiler Warnings, Adopt a Secure Coding Standard  Do not use reserved names when declaring a namespace, this practice should also be conveyed over all developing, using namespaces of any reserved variables is prohibited. Compiler warnings will be generated when attempting to override a standard or reserved namespace. By utilizing the coding standard set forth by this policy, overriding namespaces will be a nonexistent problem. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14.0 | Automatic variable checking | Checks against standard and reserved namespaces and variables. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Value-returning functions must return a value from all**  **exit paths** |
| --- | --- | --- |
| Function Returns | [STD-010-CPP] | To prevent undefined behavior from a function with a return value, a value must be returned in all instances of the function being completed. This is including exceptions that are thrown within the function. |

| **Noncompliant Code** |
| --- |
| In the instance of ‘a’ being greater than 0, a return value is not available. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| All exit paths are now covered. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques  Assertion testing and static tests will be able to detect return values from functions, whether they are given or not, and whether they are of the correct type. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.14.0 | Assertions | Assert that expected value is returned from function. |

### 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.

During the verify and test phase of the DevSecOps process is where automation of the standards in this policy should be implemented. Within that phase, static testing tools can be used to guarantee that the code being developed is in line with the standards set forth by this policy. Also, during the build phase of the process developers can adhere to best practices outlined in this policy as well to mitigate time spent debugging after the automation process. While the transition and health check phase is being commenced, security vulnerabilities can be reassessed as to make sure no new exploits were introduced in the debugging process. Lastly, the monitor and detect phase should reinforce that the automation of static testing was successful and that preventable bugs were removed in the process.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004CPP | High | Likely | Medium | P15 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Medium | Unlikely | Low | P3 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-010-CPP | Medium | Probable | Medium | P8 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest applies to data that is not currently being, rather being stored in a server or database. A secure hashing algorithm must be implemented to keep user information that is being stored within our services safe. |
| Encryption in flight | Encryption in flight would be indicative of data being transferred between a database and a network/program. This policy maintains that user information shall be encrypted during the exchange of data as to mitigate man-in-the-middle attacks. |
| Encryption in use | Encryption in use refers to data that is actively being processed from a user. For example, when a user enters their credit card information into a web-page. Encryption of the user’s card information should take place and handled with third-party services, the information should never be stored locally to the program. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users using our software must be authenticated before use. This aids in reducing the possibility of users pirating our software. Also, this ensures that the person attempting to access the software is the user stated. This can be achieved with two-step verification processes. |
| Authorization | Users should only be allowed to conduct the operations necessary for them. For example, a common user should be able to adjust read and write privileges of themself or other users in the database. Authorization should occur after the user has been authenticated. |
| Accounting | Every access to our software should be logged and monitored. The types of files accessed by users should also be logged to ensure that correct functionality for that user level is still maintaining its effectiveness. Abnormalities in login behaviors should be flagged, as it could prevent a possible brute-force attack. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/23/2024 | Added 10 Principles and Coding Standards | Michael Smith |  |
| 2.0 | 06/15/2024 | Security Policy Completion | Michael Smith |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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