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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

[C/C++ Ten Coding Standards 3](#_3dy6vkm)

[Coding Standard 1 4](#_1t3h5sf)

[Coding Standard 2 5](#_4d34og8)

[Coding Standard 3 6](#_2s8eyo1)

[Coding Standard 4 7](#_17dp8vu)

[Coding Standard 5 8](#_3rdcrjn)

[Coding Standard 6 9](#_26in1rg)

[Coding Standard 7 10](#_lnxbz9)

[Coding Standard 8 11](#_35nkun2)

[Coding Standard 9 13](#_1ksv4uv)

[Coding Standard 10 14](#_44sinio)

[Defense-in-Depth Illustration 15](#_2jxsxqh)

[Project One 15](#_z337ya)

[1.](#_3j2qqm3) Revise the C/C++ Standards 15

[2.](#_1y810tw) Risk Assessment 15

[3.](#_4i7ojhp) Automated Detection 15

[4.](#_2xcytpi) Automation 15

[5.](#_1ci93xb) Summary of Risk Assessments 16

[6.](#_3whwml4) Create Policies for Encryption and Triple A 16

[7.](#_2bn6wsx) Map the Principles 17

[Audit Controls and Management 18](#_qsh70q)

[Enforcement 18](#_3as4poj)

[Exceptions Process 18](#_1pxezwc)

[Distribution 19](#_49x2ik5)

[Policy Change Control 19](#_2p2csry)

[Policy Version History 19](#_147n2zr)

[Appendix A Lookups 19](#_3o7alnk)

[Approved C/C++ Language Acronyms 19](#_23ckvvd)

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input validation is one of the easiest and strongest ways of preventing security vulnerabilities. By validating data, we are able to have a wall of defense to preemptively stop any attacks such as buffer overflow or sql injection. This can be done by verifying that the data is the correct length and by ensuring that the type of data is also correct. |
| 1. Heed Compiler Warnings | Compiler warnings are not something to be ignored. Whenever something comes up as a warning, the first thing that should be done as a developer is address it to prevent any further vulnerabilities. Although the warnings can’t catch everything, fixing any warnings will provide a strong start to security. |
| 1. Architect and Design for Security Policies | Outlining any sort of security procedure or technique is optimal for the best security possible. Doing so will allow for any immediate security flaws to be resolved during the creation of the security policy design. |
| 1. Keep It Simple | Simplicity allows for easy understanding, modifications, and updates. While it may not allow for as much versatility in certain cases, making any design implementation simple will allow for less exploits being possible. |
| 1. Default Deny | Default deny is having authorization requirements for any function of the program or computer that can impact changes to the system or program. By denying authorization by default, we can make sure that only trusted individuals will be able to access the vulnerable parts of any system or program. |
| 1. Adhere to the Principle of Least Privilege | Similar to default deny, adhering to the principle of least privilege is an approach to minimize the amount of users to only those with the required level of privilege in any given part of a device. This once again limits the amount of people capable of accessing programs to exploit any vulnerabilities. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems allows for data to be cleaned of any potential exploits and vulnerabilities before damaging the other system. This can prevent anything from scripts to something like sql injection from transmitting. |
| 1. Practice Defense in Depth | Defense in Depth is a strong, but lengthy approach to security. By having multiple defense barriers of security techniques, any potential exploit will have to bypass all of the barriers to be able to get to vulnerable parts of the program. While seemingly the most logical approach, it does tend to be expensive in terms of development and time to implement. |
| 1. Use Effective Quality Assurance Techniques | While more defense is a strong option, it is not going to do much if the defense isn’t effective. By making sure that our effective quality assurance techniques are optimal and working as intended, we are able to reduce the need for more security layers while maintaining strong security. |
| 1. Adopt a Secure Coding Standard | Secure coding standards allows for a base guideline for the developers to adhere to. This supports the security design as well as making it simpler for any developer to maintain the security along with their team. |

### 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-DT] | Consistent data typing and practices allow for developers to be on the same page and create good coding practices since they will be more familiar with the types being used. As such, using the coding standard of maintaining consistent data types is one of the easiest implementations for secure coding. |

| **Noncompliant Code** |
| --- |
| The following two examples do not work as good data type practices |
| // Noncompliant code block  int notString = "395404"; // Assigning a string to an integer variable does not work  boolean isString = "bool"; // Assigning a string to a boolean variable does not work |

| **Compliant Code** |
| --- |
| The following two examples are good data type practices |
| // compliant code block  int numExample = 395404; // Assigning a string to an integer variable  boolean goodExample = True; // Assigning a string to a boolean variable |

| **Principles(s):** ValidateInput Data: Making sure that we are using the correct data will lead to the least errors and the best secure code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| low | likely | small | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | type compatibility | multi-purpose automatic code review tool |
| PVS-Studio | 7.31 | static code analyzer | code analyzer |
| SonarLint | 10.1 | linting tool | checks for errors while coding to preemptively fix code |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DV] | Data values being assigned correctly may not seem like much at first, but they can make or break a program or the logic behind its intent. For instance, data values representing someone’s age must be in the positives. If it were to be negative, the whole program would look bad due to not making sense. As such, maintaining data values that are appropriately assigned are necessary for best coding standards. |

| **Noncompliant Code** |
| --- |
| Data value does not work for the intended purpose even if code compiles correctly |
| int age = -45; |

| **Compliant Code** |
| --- |
| Data value fits intended purpose and compiles |
| int age = 45; |

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

| **Principles(s):** Keep It Simple, validate input data  Data values being correctly assigned are integral to any stable and secure program. Keep it simple and try to approach any programming with any ideas of data types in mind. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Small | Low | 9 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| | SonarQube | | --- | | 10.1 | S2259 | SonarQube allows for type correction and type mismatches through its multiple features including recommended fixes. |
| PMD | 6.41 | | UnnecessaryConversionForToString | | --- |  |  | | --- | | A tool for finding and fixing any incorrect string conversion in java code. |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-SC] | Ensuring that string correctness is maintained is a great standard for preventing any sort of vulnerabilities. If string correctness is not maintained, possible attacks include buffer overflow as well as sql injection. |

| **Noncompliant Code** |
| --- |
| Improper conversion |
| std::u16string utf16 = convert.from\_bytes(text); // Incorrect conversion and has potential data loss issues |

| **Compliant Code** |
| --- |
| Correct conversion |
| std::string utf8 = convert.to\_bytes(text); |

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

| **Principles(s):** Heed compiler warnings It can be very easy to fix any issues related to strings if we heed compiler warnings. Use this tool as much as you can and you will have correctness more often in your strings! |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | very likely | small | high | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| | CodeSonar | | --- | | 5.7 | STRING\_CORRECTNESS | A tool for finding any string correctness issues and helps fixing them through recommendation features. Detects string vulnerabilities. |
| | FindBugs | | --- | | 3.0.1 | | DM\_CONVERT\_CASE | | --- |  |  | | --- | | A tool that will notify the user whenever string related issues involving case-sensitive issues come up. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQLI] | SQL injection is a very severe security risk, one of which enables attackers to manipulate SQL queries and gain unauthorized access to database information. Preventing SQL injection is crucial for safeguarding the security and integrity of both the database and the application. |

| **Noncompliant Code** |
| --- |
| Improper code can lead to sql injection vulnerabilities such as the following code. |
| std::string query = "INSERT INTO users (name, email) VALUES ('" + name + "', '" + email + "');";  stmt.execute(query); |

| **Compliant Code** |
| --- |
| Proper code will reduce the sql injection from being possible and make the code more secure such as the following. |
| sql::goodExample\* pstmt = conn->goodExample("INSERT INTO users (name, email) VALUES (?, ?)");  pstmt->setString(1, name);  pstmt->setString(2, email);  pstmt->executeUpdate(); |

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

| **Principles(s):** Adopt a Secure Coding Standard, ValidateInput Data  SQL injection is one of the most common forms of attacks when it comes to code. Maintaining a secure coding standard and validating input data are essential in preventing any sql injection attempts from being successful. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| | SonarQube | | --- | | 10.1 | S3649 | prevents and detects vulnerabilities related to sql injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MP] | A lot of exploits such as buffer overflow and memory leaks are huge vulnerabilities if memory protection is not in place. Thus, making sure that memory is properly allocated, deallocated, and programming in a way that keeps memory in mind will be the best approach to safe coding. |

| **Noncompliant Code** |
| --- |
| Incorrect buffer can lead to exploits |
| char buffer[1030]; |

| **Compliant Code** |
| --- |
| Correct buffer will reduce exploit possibilities |
| std::vector<char> buffer(1030); |

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

| **Principles(s):** Practice Defense in Depth, Adopt a Secure Coding Standard  Defense in depth is useful in this case due to how vital memory and data is to any program. Having a secure coding standard along with multiple walls of defense are great approaches to securing any code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | Medium | 7 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.1 | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASSR] | Assertions are an easy way to test that the code fits the intended purpose by testing it during runtime. Assertions allow us to debug while also verifying for correctness. |

| **Noncompliant Code** |
| --- |
| Does not use assert |
| if (number < 0) {  throw std::out\_of\_range("The number cannot be less than 0");  } |

| **Compliant Code** |
| --- |
| Uses assert |
| assert(num >= 0); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques  Quality of security in code can be hard to judge without doing tests. Assertions are one of the many ways to test and see how code functions under certain conditions, thus allowing us to strengthen the security of our code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | N/A | Medium | High | 6 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | S2699 | Detects any issues related to assertions through analyzing error handling tools. |
| Coverity | 2023.06 | ASSERT\_SIDE\_EFFECT | Identifies any issues related to assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | Exception handling is a strong tool to use to catch any errors and rectify them. Using try-catch blocks, exception handling can catch a wide variety of errors and vulnerabilities to prevent them from happening. |

| **Noncompliant Code** |
| --- |
| not exception handling |
| int result = divide(a, b); |

| **Compliant Code** |
| --- |
| correct use of try-catch exception handling |
| try {  int result = divide(a, b);  } catch (ArithmeticException e) {  Logger.log(e.getMessage(), e);  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques  Similar to assertions, exceptions are another great way to test the security of our code. This is a quality assurance technique that can be used extensively in our code to make sure it works correctly if there’s ever an error. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | N/A | Medium | Medium | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | S2699 | Detects issues related to exceptions and error handling |
| Coverity | 2023.06 | EXCEPT\_SIDE\_EFFECT | Identifies any issues related to assertions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Immutable Objects | [STD-008-IMO] | Immutable objects are good coding practice as they offer a rather consistent form of data handling. By having the object be immutable, working and handling the object will be consistent and make it easier to keep secure. |

| **Noncompliant Code** |
| --- |
| not an immutable object |
| public int x; |

| **Compliant Code** |
| --- |
| immutable object |
| public final int x; |

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

| **Principles(s):**Use Effective Quality Assurance Techniques, Adopt a secure coding standard  Immutable objects allow for quality assurance and secure coding in the sense that they stabilize data handling. By having consistent form, we can keep data secure and handle all the immutable objects in the same way. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | Low | 10 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | S3604 | Verifies if objects remain immutable and notifies of issues related to said objects |
| PMD | 7.4 | ImmutableField | Checks for any objects that can be made immutable and notifies |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Don’t repeat yourself | [STD-009-DRY] | Repeating anything unnecessarily in your code can open up the code to more vulnerabilities that would not be possible otherwise. By reducing this unneeded repetition in code, you can stop your code from having more ways to be exploited. |

| **Noncompliant Code** |
| --- |
| Unnecessary repitition |
| if (username.isEmpty()) {  showError("Username cannot be empty");  }  if (password.isEmpty()) {  showError("Password cannot be empty");  }  if (email.isEmpty()) {  showError("Email cannot be empty");  } |

| **Compliant Code** |
| --- |
| Good use of no repetition |
| validateField(username, "Username");  validateField(password, "Password");  validateField(email, "Email");  void validateField(String field, String fieldName) {  if (field.isEmpty()) {  showError(fieldName + " cannot be empty");  }  } |

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

| **Principles(s):** Architect and Design for Security Policies, keep it simple, Adopt a Secure Coding Standard  Repetition is a bad habit to start as a programmer trying to have secure code, so adopting coding standards to support our endeavor in achieving more secure code as well as planning around security policies is fundamental for secure programming. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | S4144 | Checks code for unnecessary repetition and redundancies |
| PMD | 7.5 | DuplicateCode | Checks code for unnecessary repetition and redundancies |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Good Formatting | [STD-010-FRM] | Using poor formatting can lead to code issues either through the code itself or from misinterpretation of the code due to the formatting itself. Thus, by formatting the code in a way that is both easy to understand and view, the code can become more secure. |

| **Noncompliant Code** |
| --- |
| bad formatting |
| if(condition){  doSomething();  }else{  doSomethingElse();  } |

| **Compliant Code** |
| --- |
| good formatting |
| if (condition) {  doSomething();  } else {  doSomethingElse();  } |

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

| **Principles(s):** Adopt a Secure Coding Standard, Keep It Simple  Approaching code with good formatting will allow for you to easily see exactly what goes on in your code. Thus, we will be able to catch any errors in our code with more ease due to being able to tell what is happening. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Moderate | Medium | High | 8 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.5 | S3457 | Checks for formatting issues and recommends changes for fixing code. |

### 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

Integrating automation into any secure coding design will make any security policy increasingly safer. Detecting any problems and dealing with them can be tedious and difficult, so having a process to fix issues on its own without the need for intervention makes the process faster and even cheaper if it means saving a lot of time that would otherwise involve multiple humans. For automation, we can use a variety of automated tools such as SonarQube which can be used for even greater inspection of the code and its quality of security. With such a tool, we can automate code quality analysis, coding standards, and many other automated features. Automation can be implemented at any level of the defense in depth illustration, making it an essential feature in any defense in depth approach to secure coding.



While DevOps may already be established at Green pace, there are always ways to improve. This can be done by approaching secure coding policies in the mindset of DevSecOps. Secure coding is something that should be considered before planning, while designing, and even during production. Before assessing and planning, the best approach is to think about how to add secure coding policies to the design. Once that is done and you move onto the designing and building phase, you can implement and add any secure coding features to improve the code. Next, the verification and testing phase is where we can start using quality assurance techniques such as assertions and exceptions to allow us to strengthen our codebase. After we finally move into the production phase, it is important to maintain and update any security features to keep the code as secure as possible. If the code does not meet the policy standards, then it is imperative that we fix the issues through either automated correction tools or other quality assurance tools. Maintaining, responding, and stabilizing the code to stay up to par with the secure coding policies of Green Pace are a necessity.

### 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 | Unlikely | Medium | High | 2 |
| [STD-002-DV] | low | likely | small | High | 3 |
| [STD-003-SC] | high | very likely | small | high | 4 |
| [STD-004-SQLI] | Very High | Likely | medium | medium | 2 |
| [STD-005-MP] | High | Unlikely | High | Medium | 7 |
| [STD-006-ASSR] | Low | N/A | Medium | High | 6 |
| [STD-007-EXC] | Medium | N/A | Medium | Medium | 5 |
| [STD-008-IMO] | Low | Low | Low | Low | 10 |
| [STD-009-DRY] | Low | Low | Low | Low | 1 |
| [STD-010-FRM] | High | Moderate | Medium | High | 8 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 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 involves encrypting data in physical storage devices such as hard drives or databases, or similar devices such as computers and phones. Encryption of this data can be done through a multitude of encryption tools or encryption on the device itself. |
| Encryption in flight | In flight data is data that is moving from one device or database to another. To encrypt this sort of data, the best approaches are through implementing encryption features and tools into the network or servers themselves, email encryption, or authorization and authentication features. |
| Encryption in use | Encryption for data in use is what is used for programs that are constantly running such as any software or tools. Encryption tools and user authentication are two of the best approaches for maintaining data encryption for data in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of ensuring that the user is someone who should have access to the data they are trying to access. This is one of many ways of applying defense and is one of the less expensive approaches requiring only a database and server if online for user authentication. This is done at the time that the user logins and can be done when they change databases as well. |
| Authorization | Authorization is the process of granting the correct privileges to a user. To access sensitive data, authorizing after authenticating is one of the best approaches to adding layers of security. While they may have access to one database, authorization can prevent a user from viewing another database if they are not supposed to see it. This allows us to set levels of access to the user database and make for a more secure workplace. Whenever a new user is added, authorization levels by default should be low in comparison to someone who has already been granted higher levels of authorization. |
| Accounting | Accounting is the process of monitoring any user in a server or database regardless of authentication or authorization. This is an additional layer on top of the two previously mentioned to be able to see where any issues come from and stop malicious attempts before they happen. |

**\***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 | 5/25/2023 | Milestone Update | Juan S. | Prof. Conlan |
| 1.2 | 6/15/2024 | Project 1 | Juan S. | Prof. Conlan |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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