**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 | Input can involve anything from what a user inputs, or even what gets passed as a parameter through a function. Validation should be based on certain considerations, things like SQL injection, runtime errors, compile errors, and ultimately the input should match whatever the function is trying to output. |
| 1. Heed Compiler Warnings | Code should be compiled at the highest warning level based on the compiler. Static/analysis tools can be used to automatically detect/remove security flaws. |
| 1. Architect and Design for Security Policies | The software should be designed to always take security into consideration. Any feature that’s implemented, should always enforce security policies. |
| 1. Keep It Simple | The simpler code that gets written, the less likely issues will occur, and the easier it is to update later down the line if something goes wrong. Don’t over-engineer systems with too many abstract layers, and don’t write functions or classes that are too large. Ideally use SOLID principles when you can. |
| 1. Default Deny | Access should be based on a permission system, rather than anything else. This makes it so that access is always denied by default, and it’s just easier to control permission systems all around. |
| 1. Adhere to the Principle of Least Privilege | Anything that executes should require the least number of permissions necessary to get the job done. |
| 1. Sanitize Data Sent to Other Systems | All data sent to other systems should be sanitized. These are systems like databases, other machines, etc. |
| 1. Practice Defense in Depth | In a nutshell, this is just adding layers of defense to the system. It’s sort of like 2FA, where in that case if a password is cracked, the attacker still must get through the 2FA system. And depending on how sensitive the information is, more layers may need to be added beyond just 2. |
| 1. Use Effective Quality Assurance Techniques | Using a QA system to test/penetrate the system is always a great layer to add. A lot of things can be caught with this, rather than just always pushing the code to master. I think the best method here for changes is QA testing, and a pull request. |
| 1. Adopt a Secure Coding Standard | This one is pretty much in the name, but a defined coding standard is a must. Not just for security, but also for continuity. The entire code base needs to be mostly the same in terms of design patterns, regardless of how many developers interact with it. |

### 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-C++] | This is important so that the code can be consistent, clear, and safe. This ensures variables and functions have predictable behavior. |

| **Noncompliant Code** |
| --- |
| “auto num = 10;" |
| It’s not obvious what “num” is as it’s being typed in a loose manner. It would be better to do something like “int num = 10;” or “double num = 10;” to be sure the code is clear. |

| **Compliant Code** |
| --- |
| “int num = 10;” |
| It’s very clear what the data type of num is, there’s no getting this confused. |

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

| **Principles(s):** Clarity and Predictability. This is largely for clear concise code, not chaos. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | Modernize-use-auto | Suggests explicit data over auto |
| Cppcheck | LATEST | explicitTypeCheck | Suggests proper data types |
| SonarQube | LATEST | LangTypeGuard | Makes sure data types are consistent across the code base |
| GCC | LATEST | -Wall | Highlights warnings including ambiguous data types |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-C++] | Its important variables are properly instantiated before being used. This prevents unknown behavior, and weird bugs. |

| **Noncompliant Code** |
| --- |
| “int num;” |
| Here num isn’t being defined, which can be a problem if it’s attempted to be used. Even if it were not defined, and defined through come complex means (like if statements or for loops), this still isn’t great. |

| **Compliant Code** |
| --- |
| “int num = 0;” |
| This solves the issue, and allows the num variable to be used right away. |

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

| **Principles(s):** Initialize Safety. This forces all variables to always be initialized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | Check-value-init | Checks for non-initialization |
| Cppcheck | LATEST | uninitializedVariable | Checks for non-initialization |
| SonarQube | LATEST | CppValueCheck | Detects and warns about non initialized values |
| GCC | LATEST | Wuinitialized | Shows warnings for non-initialized variables |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-C++] | This ensures string manipulation is done safely and correctly, and prevents common string issues such as buffer overflows. |

| **Noncompliant Code** |
| --- |
| char str[10];  strcpy(str, “This string is way too long for this buffer.”); |
| A copy is attempting to be made, but the string is too long, which will result in a buffer overflow. |

| **Compliant Code** |
| --- |
| std::string str = “Beautiful safe string” |
| std::string is being used, which manages memory automatically and avoids the buffer overflow issue. No reason to work on a low level if you don’t have to. |

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

| **Principles(s):** [Memory Safety: Ensures correct string operations] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Often | High | Urgent | Level 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | check-strcpy-safety | Checks for buffer overflows when copying strings. |
| Cppcheck | LATEST | stringOverflow | Identifies buffer overflows in string operations. |
| SonarQube | LATEST | CppStringSafety | Analyzes string operations for pitfalls. |
| GCC | LATEST | -Wformat-security | Highlights unsafe string operations which lead to vulnerabilities. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C++] | This one is completely necessary as it can avoid devastating SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| SELECT \* FROM users WHERE name = ‘%name\_variable%’; |
| This is allowing the user to directly interact with the database based on the input of their username. |

| **Compliant Code** |
| --- |
| std::string username = user.getName();  //...  sqlite3\_stmt \*stmt;  std::string query = "SELECT \* FROM users WHERE name = ?";  sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, 0);  sqlite3\_bind\_text(stmt, 1, userName.c\_str(), -1, SQLITE\_STATIC); |
| This is a prepared statement, which makes injection impossible. |

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

| **Principles(s):** Data Integrity & Security. Using prepared statements is crucial with database queries as it helps avoid SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | sql-injection-check | Highlights SQL injections |
| Cppcheck | LATEST | sqlSafetyCheck | Identifies unsafe SQL queries |
| SonarQube | LATEST | CppSqlInjectionDetect | Detects possible SQL injection |
| Flawfinder | LATEST | --dataonly | Scans code for security vulnerabilities, including SQL injections. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C++] | This will help standardize memory relates issues, such as leaks, dangling pointers, and overflows. |

| **Noncompliant Code** |
| --- |
| char\* buffer = new char[10]; |
| It’s not the statement that’s wrong here, it’s the lack of deleting the memory which is the issue. |

| **Compliant Code** |
| --- |
| char\* buffer = new char[10];  delete[] buffer; |
| Here the memory is being deleted correctly. |

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

| **Principles(s):** Memory Management & Safety. Properly handling the how memory gets cleaned up, which helps avoid potential memory leaks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Frequent | Moderate | Critical | Level 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | check-memory-leak | Identifies memory leaks |
| Cppcheck | LATEST | memleak | Detects memory leaks due to missing delete or delete[]. |
| SonarQube | LATEST | CppMemProtection | Detects unsafe memory operations |
| Valgrind | LATEST | Memcheck | Analyzes memory usage and detects leaks |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C++] | This will help detect errors and strange conditions before the actual execution of the code. This makes debugging a whole lot easier. |

| **Noncompliant Code** |
| --- |
| int denom = 0;  double result = 10 / denom; |
| This is a possible divide by 0 situation, which we all know cannot happen. |

| **Compliant Code** |
| --- |
| int denom = 0;  assert(denom != 0 && "Division by zero error");  double result = 10 / denom; |
| This is just checking for 0. Personally I think adding random asserts is really good for testing, but not good for production code. I think an actual check makes more sense. |

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

| **Principles(s):** Error Detection & Robustness: Provides feedback during testing and development |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | bugprone-assert-side-effect | Checks for assertions that have side effects. |
| Cppcheck | LATEST | assertWithSideEffect | Warns if an assert has a side effect. |
| SonarQube | LATEST | CppAssertionEffect | Detects any unintended effects in assertions. |
| Static Asserts Checker | LATEST | CheckAsserts | Checks for incorrect usage of static\_assert |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-C++] | Coding exceptions throughout the software is necessary so that all the errors can be properly handled. Non error handled code makes debugging a literal living nightmare. |

| **Noncompliant Code** |
| --- |
| std::vector<int> numbers;  //... (some code)  int val = numbers.at(10); // This will throw std::out\_of\_range if the vector size is less than 11. |
| The code isn’t catching the out-of-range issue here. |

| **Compliant Code** |
| --- |
| std::vector<int> numbers;  //... (some code)  try {  int val = numbers.at(10);  }  catch (const std::out\_of\_range& e) {  std::cerr << "Out of Range error: " << e.what() << '\n';  } |
| Same example, but now the error is properly being caught. This makes debugging runtime issues like this, super easy to find and fix. Though I do think printing the entire exception is also nice for debugging. But maybe that’s just me. |

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

| **Principles(s):** Error Handling & Robustness: Ensures that the software properly handles errors, and doesn't have any runtime errors |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | LATEST | bugprone-exception-escape | Checks for good exceptions |
| Cppcheck | LATEST | exceptionSafety | Warns about code that might have undefined behavior because an exception is thrown. |
| SonarQube | LATEST | CppExceptionSafety | Ensures that functions are exception-safe. |
| GCC Compiler | LATEST | -fexceptions | Generates additional code needed by the exception handling runtime. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Function Sizes | [STD-008-C++] | The size of a function matters, and the larger a function is, the less obvious it becomes. This is one of those situations where only the person who wrote it understands it, and they won’t even be able to a few months later after not looking at it. |

| **Noncompliant Code** |
| --- |
| void runGame() {  // 500 lines of code here  } |
| This is bad because it simply cannot be used by other developers, it’s not updatable, it’s not maintainable. It’s like having a dresser droor, and shoving your clothes, laptop, food, and everything you own in it. Good luck ever finding anything lol. |

| **Compliant Code** |
| --- |
| void initiateGameLoop() {  }  void loadMaps() {  }  // etc |
| This is following the “one thing” principle as well, as each function as one job, and should only ever do one thing. |

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

| **Principles(s):** Single Responsibility Principle: Ensures clean and reusable code throughout the code base |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | LATEST | TooLongFunction | Check for functions that have grown too long or complex. |
| Cppcheck | LATEST | functionLength | Warns about functions that have a large number of lines. |
| Clang-Tidy | LATEST | readability-function-size | Enforces limits on function length and complexity. |
| Codacy | LATEST | ComplexFunctions | Warns about functions that exceed a specified line count or complexity threshold. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Naming Conventions | [STD-009-C++] | This is just a readability issue and has nothing to do with the computers ability to compile the code. |

| **Noncompliant Code** |
| --- |
| Int xyz = 5;  float bbyz2 = 35;  void abc() {  } |
| This is an extreme case, but it’s not uncommon to see things like “int x” or something of that nature. This is bad because it cannot be easily understood by anyone other than the creator. This forces the developer to understand all of the code, rather than just simply reading it at a top level. |

| **Compliant Code** |
| --- |
| Int dogAge = 5;  float teacherAge = 35;  void giveTeacherAFreeDog() {  // Bark bark  } |
| Now the code can be understood without having to really read it at a low level. |

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

| **Principles(s):** Self-documenting Code Principle: Ensuring the code is readable, and doesn't just rely on sloppy comments |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | LATEST | VariableNamingConventions | Checks that variables, constants, and functions are named according to predefined conventions. |
| Cppcheck | LATEST | NamingConvention | Highlights variables and functions that don't follow the project's naming standards |
| Clang-Tidy | LATEST | readability-identifier-naming | Enforces naming conventions on various kinds of identifiers. |
| Codacy | LATEST | NamingPatterns | Ensures variables, functions, and classes have specific naming patterns |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Use of Constants | [STD-010-C++] | The idea here is to just re-use code if it’s going to exist in multiple places throughout the software. This is a tradeoff of space/time, but the initial space for storing long term variables is generally worth it. Especially for updatability. |

| **Noncompliant Code** |
| --- |
| double calculateArea() {  double radius = 5.0;  return 3.14159 \* radius \* radius; // '3.14159' is a magic number  } |
| This doesn’t have the PI value stored forever, but it is re-created every time this is called, and if PI was used in 50 places through the code randomly, and for whatever reason needed to be updated. Lol good luck have fun. |

| **Compliant Code** |
| --- |
| const double PI = 3.14159;  double calculateArea() {  double radius = 5.0;  return PI \* radius \* radius; // Use the defined constant 'PI'  } |
| This improves readability, usability, and updatability. It’s a win win win. |

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

| **Principles(s):** Principle of DRY (Don't Repeat Yourself): No duplication code! NEVER! Well sometimes, but let’s try to avoid it |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Checker | LATEST | ConstantUsageCheck | Checks for the use of constants in the code. |
| Code Quality Tool | LATEST | RepeatedValueConsistencyCheck | Verifies consistency of repeated values across the codebase. |
| Magic Number Detector | LATEST | MagicNumberCheck | Detects and reports magic numbers without named constants. |
| Refactoring Assistant | LATEST | ConstantSuggestor | Suggests areas where constants can be introduced for repeated values. |

### 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 with 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.

Automation in this can and should be used wherever possible, specifically meaning any specific tasks that a human doesn’t need to perform and can assist any human interacting with the software system. I think the best approach for this company is to just update the IDEs everyone is using, and maybe install a few automation tools that can help. I think this is a great first step and avoids disrupting too much of what is already there, which of course would be a bad thing.

### 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 | Medium | Unlikely | Low | Moderate | 2 |
| STD-002-CPP | Medium | Low | Low | Medium | 2 |
| STD-003-CPP | Critical | Often | High | Urgent | Level 1 |
| STD-004-CPP | Critical | High | High | Urgent | Level 1 |
| STD-005-CPP | High | Frequent | Moderate | Critical | Level 2 |
| STD-006-CPP | Moderate | Frequent | Low | Medium | 3 |
| STD-007-CPP | High | Frequent | Medium | High | 4 |
| STD-008-CPP | Low | Frequent | High | Medium | 3 |
| STD-009-CPP | Low | High | Low | Low | 1 |
| STD-0010-CPP | Medium | Low | Low | Medium | 1 |

### Create Policies for Encryption and Triple A

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

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

Write policies for each and explain what they are, how they 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 | This is encryption that is stored, but not being used. Something like a hard drive, or maybe even a USB stick. This is specifically for long term storage though, so probably not a USB stick honestly. But maybe? |
| Encryption at flight | This is encryption in transit, AKA being moved from one place to another, or from one machine to another machine. Common technologies here are things like HTTPS, SSL, and TLS |
| Encryption in use | This protects the data while it’s being used/processed. This is very important in cloud environments, where data is likely being shared on insecure systems. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying the identify of a user, device, or system. This is usually done when an attempt to access the system is made. |
| Authorization | Determines the authorization of whatever actor is attempting to connect to the system. |
| Accounting | This is the process of tracking user activities and resource usage, AKA this just logging in a nutshell. If everything gets locked and tracked, everything that happens in the system can be determined. |

**\***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 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 always follow this policy.

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 the 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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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