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



# Green Pace Secure Development Policy

# Jonathan Wolanyk

# Professor Prasad

# CS-405: Secure Coding

# March 18th, 2023

## 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 validation involves ensuring that data that a program receives consists of the correct format and falls within an acceptable range. Common types of attacks, such as SQL injection attacks, are prevented with such validation. User input validation forces programs to behave as intended to reduce the likelihood of vulnerabilities being exploited by users. |
| 1. Heed Compiler Warnings | When a compiler issues a warning after attempting to compile a program, it is issuing a detection notification of a potential problem with the code. It is important that programmers do not ignore such warnings since this could result in security vulnerabilities in the code. |
| 1. Architect and Design for Security Policies | Considering security requirements while designing a program results in more secure programs overall. While designing programs, the security policies are meant to help identify potential threats and vulnerabilities before the program is launched. |
| 1. Keep It Simple | When unnecessary complexity is added to a program, it becomes more difficult to test and secure. When a code base is kept simple, developers can easily maintain, understand, and test the program to monitor and address vulnerabilities. |
| 1. Default Deny | Denying all access to a program by default is an important practice to maintain a program’s integrity. This principle involves denying all access requests unless the requestor is explicitly allowed to receive access. This forces users to prove they should be able to access the program, rather than the program having to justify granting access to each user. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the principle of least privilege involves giving users the least amount of access needed to perform intended tasks on a program. This practice reduces the likelihood of a user being able to access information that should not be accessible for their role in the program. For instance, a general user should have the privileges to view their own information, but not the information of other users. |
| 1. Sanitize Data Sent to Other Systems | It is important to ensure that data being sent out of a system does not compromise the integrity of the system itself. Since the system is unsure of the connection on the other end (the data could be received on an unsecure network), it is important to protect sensitive data as it leaves the system to avoid unauthorized access to such information. |
| 1. Practice Defense in Depth | Defense in depth involves utilizing multiple layers of security to protect against attacks. Similar to multiple layers of a wall, the defense in depth philosophy puts forth that an unauthorized user may get through one layer of defense, but would be unable to penetrate every layer of the security system if implemented correctly. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance involves testing and validating code to ensure it functions as expected. Quality assurance techniques should be implemented to ensure integrity of a system from a security standpoint as well. All systems should use manual, automation, unit, integration, and penetration testing to identify and prevent potential attacks. |
| 1. Adopt a Secure Coding Standard | Secure coding standards are a set of guidelines and best practices to ensure a system is created in a way that prevents as many security vulnerabilities as possible. Such standards allow for a consistent approach to system development as well. Adoption of secure coding standards ensures that a programming team creates a system in a secure manner while following industry and company compliance requirements. |

#### 

#### Coding Standard 1

| **Coding Standard** | **Label** | **Include the appropriate type information in function declarators** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Function declarators must be declared with the appropriate type information, including a return type and parameter list. If type information is not properly specified in a function declarator, the compiler cannot properly check function type information. When using standard library calls, the easiest (and preferred) way to obtain function declarators with appropriate type information is to include the appropriate header file. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the identifier-list form for parameter declarations: |
| **int** max(a, b)  **int** a, b;  {  **return** a > b ? a : b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, int is the type specifier, max(int a, int b) is the function declarator, and the block within the curly braces is the function body: |
| **int** max(**int** a, **int** b) {  **return** a > b ? a : b;  } |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not begin integer constants with 0 when specifying a decimal value** |
| --- | --- | --- |
| **Data Value** | [STD-002- CPP] | The C Standard defines octal constants as a 0 followed by octal digits (0 1 2 3 4 5 6 7). Programming errors can occur when decimal values are mistakenly specified as octal constants. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a decimal constant is mistakenly prefaced with zeros so that all the constants are a fixed length: |
| i\_array[0] = 2719;  i\_array[1] = 4435;  i\_array[2] = 0042; |

| **Compliant Code** |
| --- |
| To avoid using wrong values and to make the code more readable, do not preface constants with zeroes if the value is meant to be decimal: |
| i\_array[0] = 2719;  i\_array[1] = 4435;  i\_array[2] =   42; |

#### 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] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ std::basic\_string template requires no such character. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    **void** f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004- CPP] | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted. |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() [Viega 2003]: |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Clear sensitive information stored in reusable resources** |
| --- | --- | --- |
| **Memory Protection** | [STD-005- CPP] | Sensitive data stored in reusable resources may be inadvertently leaked to a less privileged user or attacker if not properly cleared. Examples of reusable resources include the following; dynamically allocated memory, statically allocated memory, automatically allocated (stack) memory, memory caches, disk, disk caches .The manner in which sensitive information can be properly cleared varies depending on the resource type and platform. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, sensitive information stored in the dynamically allocated memory referenced by secret is copied to the dynamically allocated buffer, new\_secret, which is processed and eventually deallocated by a call to free(). Because the memory is not cleared, it may be reallocated to another section of the program where the information stored in new\_secret may be unintentionally leaked. |
| **char** \*secret;  /\* Initialize secret to a null-terminated byte string,     of less than SIZE\_MAX chars \*/    **size\_t** size = **strlen**(secret);  **char** \*new\_secret;  new\_secret = (**char** \*)**malloc**(size+1);  **if** (!new\_secret) {    /\* Handle error \*/  }  **strcpy**(new\_secret, secret);    /\* Process new\_secret... \*/    **free**(new\_secret);  new\_secret = NULL; |

| **Compliant Code** |
| --- |
| To prevent information leakage, dynamic memory containing sensitive information should be sanitized before being freed. Sanitization is commonly accomplished by clearing the allocated space (that is, filling the space with '\0' characters). |
| **char** \*secret;  /\* Initialize secret to a null-terminated byte string,     of less than SIZE\_MAX chars \*/    **size\_t** size = **strlen**(secret);  **char** \*new\_secret;  /\* Use calloc() to zero-out allocated space \*/  new\_secret = (**char** \*)**calloc**(size+1, **sizeof**(**char**));  **if** (!new\_secret) {    /\* Handle error \*/  }  **strcpy**(new\_secret, secret);    /\* Process new\_secret... \*/    /\* Sanitize memory \*/  memset\_s(new\_secret, '\0', size);  **free**(new\_secret);  new\_secret = NULL; |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006- CPP] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see MSC11-C. Incorporate diagnostic tests using assertions). The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007- CPP] | All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. As per ERR50-CPP-EX1, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the main() function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Be careful using functions that use file names for identification** |
| --- | --- | --- |
| **Function Naming Conventions** | [STD-008- CPP] | Many file-related security vulnerabilities result from a program accessing an unintended file object because file names are only loosely bound to underlying file objects. File names provide no information regarding the nature of the file object itself. Furthermore, the binding of a file name to a file object is reasserted every time the file name is used in an operation. File descriptors and FILE pointers are bound to underlying file objects by the operating system. (See FIO03-C. Do not make assumptions about fopen() and file creation.) Accessing files via file descriptors or FILE pointers rather than file names provides a greater degree of certainty as to which object is acted upon. It is recommended that files be accessed through file descriptors or FILE pointers where possible. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function chmod() is called to set the permissions of a file. However, it is not clear whether the file object referred to by file\_name refers to the same object in the call to fopen() and in the call to chmod(). |
| **char** \*file\_name;  **FILE** \*f\_ptr;    /\* Initialize file\_name \*/    f\_ptr = **fopen**(file\_name, "w");  **if** (f\_ptr == NULL)  {    /\* Handle error \*/  }    /\* ... \*/    **if** (chmod(file\_name, S\_IRUSR) == -1) {    /\* Handle error \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the POSIX fchmod() and open() functions [IEEE Std 1003.1:2013]. Using these functions guarantees that the file opened is the same file that is operated on. |
| **char** \*file\_name;  **int** fd;    /\* Initialize file\_name \*/    fd = open(    file\_name,    O\_WRONLY | O\_CREAT | O\_EXCL,    S\_IRWXU  );  **if** (fd == -1) {    /\* Handle error \*/  }    /\* ... \*/    **if** (fchmod(fd, S\_IRUSR) == -1) {    /\* Handle error \*/  } |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Functions should validate their parameters** |
| --- | --- | --- |
| **Parameter Validation** | [STD-009- CPP] | Redundant testing by caller and by callee as a style of defensive programming is largely discredited in the C and C++ communities, the main problem being performance. The usual discipline in C and C++ is to require validation on only one side of each interface. Requiring the caller to validate arguments can result in faster code because the caller may understand certain invariants that prevent invalid values from being passed. Requiring the callee to validate arguments allows the validation code to be encapsulated in one location, reducing the size of the code and making it more likely that these checks are performed in a consistent and correct fashion. For safety and security reasons, this standard recommends that the called function validate its parameters. Validity checks allow the function to survive at least some forms of improper usage, enabling an application using the function to likewise survive. Validity checks can also simplify the task of determining the condition that caused the invalid parameter. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, setfile() and usefile() do not validate their parameters. It is possible that an invalid file pointer can be used by the library, corrupting the library's internal state and exposing a vulnerability. |
| /\* Sets some internal state in the library \*/  **extern** **int** setfile(**FILE** \*file);    /\* Performs some action using the file passed earlier \*/  **extern** **int** usefile();    **static** **FILE** \*myFile;    **void** setfile(**FILE** \*file) {      myFile = file;  }    **void** usefile(**void**) {      /\* Perform some action here \*/  } |

| **Compliant Code** |
| --- |
| Validating the function parameters and verifying the internal state leads to consistency of program execution and may eliminate potential vulnerabilities. In addition, implementing commit or rollback semantics (leaving program state unchanged on error) is a desirable practice for error safety. |
| /\* Sets some internal state in the library \*/  **extern** errno\_t setfile(**FILE** \*file);    /\* Performs some action using the file passed earlier \*/  **extern** errno\_t usefile(**void**);    **static** **FILE** \*myFile;    errno\_t setfile(**FILE** \*file) {  **if** (file && !**ferror**(file) && !**feof**(file)) {      myFile = file;  **return** 0;    }      /\* Error safety: leave myFile unchanged \*/  **return** -1;  }    errno\_t usefile(**void**) {  **if** (!myFile) **return** -1;        /\*       \* Perform other checks if needed; return       \* error condition.       \*/        /\* Perform some action here \*/  **return** 0;  } |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Ensure that sensitive data is not written out to disk** |
| --- | --- | --- |
| **Data Protection** | [STD-010- CPP] | Developers should take steps to prevent sensitive information such as passwords, cryptographic keys, and other secrets from being inadvertently leaked. Preventive measures include attempting to keep such data from being written to disk. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, sensitive information is supposedly stored in the dynamically allocated buffer, secret, which is processed and eventually cleared by a call to memset\_s(). The memory page containing secret can be swapped out to disk. If the program crashes before the call to memset\_s() completes, the information stored in secret may be stored in the core dump. |
| **char** \*secret;    secret = (**char** \*)**malloc**(size+1);  **if** (!secret) {    /\* Handle error \*/  }    /\* Perform operations using secret... \*/    memset\_s(secret, '\0', size+1);  **free**(secret);  secret = NULL; |

| **Compliant Code** |
| --- |
| To prevent the information from being written to a core dump, the size of core dumps that the program will generate should be set to 0 using setrlimit(): |
| #include <sys/resource.h>  /\* ... \*/  **struct** rlimit limit;  limit.rlim\_cur = 0;  limit.rlim\_max = 0;  **if** (setrlimit(RLIMIT\_CORE, &limit) != 0) {      /\* Handle error \*/  }    **char** \*secret;    secret = (**char** \*)**malloc**(size+1);  **if** (!secret) {    /\* Handle error \*/  }    /\* Perform operations using secret... \*/    memset\_s(secret, '\0', size+1);  **free**(secret);  secret = NULL; |

### 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.

[Insert your written explanations here.]

### 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 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | [Insert text.] |
| Encryption at flight | [Insert text.] |
| Encryption in use | [Insert text.] |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | [Insert text.] |
| Authorization | [Insert text.] |
| Accounting | [Insert text.] |

**\***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 |  |
| [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 |