

# Green Pace Secure Development Policy

# Tristin Watson

## 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 | Validating user input allows programmers to ensure that only the set parameters are allowed in their program. SQL injections are put to a near immediate stop with just validating the user input. It also restricts buffer overflows through limiting the amount of characters a user is allowed to input. |
| 1. Heed Compiler Warnings | Compiler warnings let programmers know where a potential problem might lie and fix it early. Eliminating security flaws is a must in any software program if you wish to keep it secure. |
| 1. Architect and Design for Security Policies | Design your system with security in mind. To keep your software safe, place privilege settings at different intervals and divide your system based on these privileges to ensure that your data is safe. |
| 1. Keep It Simple | Although easy on paper, keeping your software simple is a much harder task to actually do. Modularize your code and keep it organized. Messy code is a huge security flaw in and of itself. |
| 1. Default Deny | Deny access by default. This means you only need to validate the users credentials and keeps a random unknown loophole from becoming your downfall. |
| 1. Adhere to the Principle of Least Privilege | Set privileges from least to greatest in structure. This means hackers must breach each and every set of the structure to gains full access to your system and keeps what they manage to get to a minimum otherwise. |
| 1. Sanitize Data Sent to Other Systems | Any data passed through your system must be sanitized. Injection attacks are commonplace and an easy way to avoid such attacks is to identify/restrict them at the start. |
| 1. Practice Defense in Depth | Multiple defensive measures are a must have for any software. While one defense measure might be a specific flaw, the other measures do not and cover of said flaw later. This allows your overall security to be structurally sound against any common attack out there. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance is a must use in any software system. Ensuring that your code is neat and secure keep vulnerabilities to a minimum. It also allows for vulnerabilities to be found sooner rather than later when such a problem does occur. |
| 1. Adopt a Secure Coding Standard | Coding standards allow for your code to be clean and modularized. Clean code can be used later without much difficulty while messy code needs to first be fixed and error checked. Applying coding standards can keep your software secure through simple steps that allow for identifying mistakes later on. |

(safeonline, 2018)

### 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-INT] | Unsigned Integers must not wrap |

| **Noncompliant Code** |
| --- |
| The code block below can lead to a wrap since the addition of int ui\_a and int ui\_b can result in a number greater than the maximum data type value. |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This code block checks if your integers when added together surpass the data limit. |
| #include <limits.h>    **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum;  **if** (UINT\_MAX - ui\_a < ui\_b) {      /\* Handle error \*/    } **else** {      usum = ui\_a + ui\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** Validate Input. The code above ensures that if an integer can surpass the data limit, then the error is handled. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Integer Overflow | Full Check |
| Coverity | 2017.07 | Integer Overflow | Implemented |
| LDRA tool suite | 9.7.1 | 493 S, 494 S | Partial Implementation |
| TrustinSoft Analyzer | 1.38 | Unsigned Overflow | Exhaustively Verified |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-INT] | Integer Precision Correctness |

| **Noncompliant Code** |
| --- |
| While this code error checks correctly, that is only correct on a platform that uses a 64-bit unsigned integer. If the integer is 48-bit, then an error can occur. |
| #include <limits.h>    unsigned **int** pow2(unsigned **int** **exp**) {  **if** (**exp** >= **sizeof**(unsigned **int**) \* CHAR\_BIT) {      /\* Handle error \*/    }  **return** 1 << **exp**;  } |

| **Compliant Code** |
| --- |
| Popcount() allows for precision determining of integers. |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  **size\_t** popcount(uintmax\_t num) {  **size\_t** precision = 0;  **while** (num != 0) {  **if** (num % 2 == 1) {        precision++;      }      num >>= 1;    }  **return** precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s):** Input Validation. The code block above ensures that even for unique scenarios, the precision is correct. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported Astree Report |
| Code Sonar | 7.4p0 | LANG.ARITH.BIGSHIFT | Shift Exceeds Width |
| Helix QAC | 2023.3 | C0582 C++3115 | Use correct integer precision |
| Polyspace Bug Finder | R2023b | CERT C: Rule INT35-C | Checks correct integer precision |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | Do Not Modify String Literals |

| **Noncompliant Code** |
| --- |
| Modifying this string literal results in undefined behavior. |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Creating a copy of the string as an array allows for safe modification of the string. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** Secure Coding Standard. Accessing an array is much more secure than modifying a string literal. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | String-literal-modification-write-to-string-literal | Full Check |
| Axivion | 7.2.0 | CertC-STR30 | Full Implementation |
| Compass/ROSE |  |  | Simple Detection |
| Coverity | 2017.07 | PW | Deprecates convertion |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-FIO] | Valid String Format |

| **Noncompliant Code** |
| --- |
| The d specifier is incorrect matched up with error\_type. This also applies with the s specifier to error\_msg. |
| #include <stdio.h>    **void** func(**void**) {  **const** **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;    /\* ... \*/  **printf**("Error (type %s): %d\n", error\_type, error\_msg);    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The arguments conversion specifications match correctly. |
| #include <stdio.h>    **void** func(**void**) {  **const** **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;    /\* ... \*/  **printf**("Error (type %d): %s\n", error\_type, error\_msg);      /\* ... \*/  } |

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

| **Principles(s):** Secure Coding Standards. Correctly formatting strings ensures that your code is secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO47 | Full Implementation |
| CodeSonar | 7.4p0 | IO.INJ.FMT.MISC.FMT.MISCFMTTYPE | Format String Error |
| Coverity | 2017.07 | PW | Reports when arguments differ in length |
| GCC | 4.3.5 |  | Detects Violations |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | Freed Memory Must Not be Accessed |

| **Noncompliant Code** |
| --- |
| P is freed in this example before p->next is set. This means freed memory is being accessed each time the function is ran. |
| #include <stdlib.h>    **struct** node {  **int** value;  **struct** node \*next;  };    **void** free\_list(**struct** node \*head) {  **for** (**struct** node \*p = head; p != NULL; p = p->next) {  **free**(p);    }  } |

| **Compliant Code** |
| --- |
| Storing p->next in q then freeing p allows for correct memory storage. |
| #include <stdlib.h>    **struct** node {  **int** value;  **struct** node \*next;  };    **void** free\_list(**struct** node \*head) {  **struct** node \*q;  **for** (**struct** node \*p = head; p != NULL; p = q) {      q = p->next;  **free**(p);    }  } |

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

| **Principles(s):** Adopt Secure Coding Standards. Do not access freed memory since it could cause errors in your program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Dangling\_pointer\_use | Supported |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses |
| CodeSonar | 7.4p0 | ALLOC.UAF | Use after free |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Detect memory deallocations |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-ERR] | Behavior Of Assert() and Abort() Termination |

| **Noncompliant Code** |
| --- |
| If the assert() function fails in this code block, the cleanup function is never called. |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **assert**(/\* Something bad didn't happen \*/);      /\* ... \*/  } |

| **Compliant Code** |
| --- |
| An if statement replaces assert() to provide termination with error. |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **if** (/\* Something bad happened \*/) {  **exit**(EXIT\_FAILURE);    }      /\* ... \*/  } |

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

| **Principles(s):** Heed Compiler Warnings. Asserts are compiler warnings which should be addressed if flagged. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  | Can detect violations |
| LDRA tool suite | 9.7.1 | 44S | Enhanced enforcement |
| Parasoft C/C++ test | 2023.1 | CERT\_C-ERR06-a | Do not use assertions |
| PC-lint Plus | 1.4 | 586 | Full Support |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-ERR] | Handle All Exceptions |

| **Noncompliant Code** |
| --- |
| No error handling can make the program crash. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| The catch statement lets any unresolved errors be caught instead of crashing the program. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques. Handling all exceptions is integral to making your code secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Main-function-catch-all-early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 7.4p0 | LANG.STRUCT.UCTCH | Unreadable Catch |
| Helix QAC | 2023.3 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Floating Points | [STD-008-FLP] | Floating-point Should Be In Range of New Data Type |

| **Noncompliant Code** |
| --- |
| The code block below changes a float to an int without error checks. |
| **void** func(**float** f\_a) {  **int** i\_a;      /\* Undefined if the integral part of f\_a cannot be represented. \*/    i\_a = f\_a;  } |

| **Compliant Code** |
| --- |
| This code block checks if the float value lies within the range of the next data type. |
| #include <float.h>  #include <limits.h>  #include <math.h>  #include <stddef.h>  #include <stdint.h>    **extern** **size\_t** popcount(uintmax\_t); /\* See INT35-C \*/  #define PRECISION(umax\_value) popcount(umax\_value)    **void** func(**float** f\_a) {  **int** i\_a;    **if** (isnan(f\_a) ||        PRECISION(INT\_MAX) < log2f(fabsf(f\_a)) ||        (f\_a != 0.0F && fabsf(f\_a) < FLT\_MIN)) {      /\* Handle error \*/    } **else** {      i\_a = f\_a;    }  } |

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

| **Principles(s):** Input Validation. Ensuring floats are in range of new data types keeps overflows from happening. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported Astree Reports |
| CodeSonar | 7.4p0 | LANG.TYPE.IAT | Assignment Type |
| TrustinSoft Analyzer | 1.38 | Float\_to\_int | Exhaustively Verified |
| PVS-Studio | 7.26 | V615, V2003, V2004 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Indentifiers | [STD-009-DCL] | Declare Identifiers |

| **Noncompliant Code** |
| --- |
| No identifier stated. |
| **extern** foo; |

| **Compliant Code** |
| --- |
| Identifier is stated. |
| **extern** **int** foo; |

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

| **Principles(s):** Use Effective Quality Assurance Techniques. Declaring identifiers helps keep code clean. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Type-specifier | Full Check |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL31 | Full Implementation |
| Coverity | 2017.07 | MISRA C 2012 RULE 8.1 | Implemented |
| ÉCLAIR | 1.2 | CC2.DCL31 | Full Implementation |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Array Size | [STD-010-ARR] | Array Size Argument Must Be Within Range |

| **Noncompliant Code** |
| --- |
| No preset size variable allows for array to be out of range. |
| #include <stddef.h>    **extern** **void** do\_work(**int** \*array, **size\_t** size);    **void** func(**size\_t** size) {  **int** vla[size];    do\_work(vla, size);  } |

| **Compliant Code** |
| --- |
| Setting a valid range for size during input keeps the array in range. |
| #include <stdint.h>  #include <stdlib.h>    **enum** { MAX\_ARRAY = 1024 };  **extern** **void** do\_work(**int** \*array, **size\_t** size);    **void** func(**size\_t** size) {  **if** (0 == size || SIZE\_MAX / **sizeof**(**int**) < size) {      /\* Handle error \*/  **return**;    }  **if** (size < MAX\_ARRAY) {  **int** vla[size];      do\_work(vla, size);    } **else** {  **int** \*array = (**int** \*)**malloc**(size \* **sizeof**(**int**));  **if** (array == NULL) {        /\* Handle error \*/      }      do\_work(array, size);  **free**(array);    }  } |

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

| **Principles(s):** Input Validation. The code block above handles errors if the array size will surpass the specified range. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | ALLOC.SIZE.IOFLOW | Integer Overflow Size |
| Coverity | 2017.07 | REVERSE\_NEGATIVE | Full Implementation |
| Helix QAC | 2023.3 | C1051 |  |
| LDRA tool suite | 9.7.1 | 621 S | Enhanced enforcement |

(sei.cmu)

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

The automation and security of a program should be done during the entire cycle of production. This includes pre-production since planning for a secure program is a must do process. There are numerous ways to develop a secure program but doing it during production allows for you to save money, time, and also ensures there are not any forgotten and unsecured items in the program.

### 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-INT | Low | Unlikely | Medium | P2 | L3 |
| STD-003-STR | Low | Likely | Low | P9 | L2 |
| STD-004-FIO | High | Unlikely | Medium | P6 | L2 |
| STD-005-MEM | High | Likely | Medium | P18 | L1 |
| STD-006-ERR | Medium | Unlikely | Medium | P4 | L3 |
| STD-007-ERR | Low | Probable | Medium | P4 | L3 |
| STD-008-FLP | Low | Unlikely | Low | P3 | L3 |
| STD-009-DCL | Low | Unlikely | Low | P3 | L3 |
| STD-010-ARR | High | Probable | High | P6 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Used to securing data stored on a physical disk. Encrypting data stored on a disk keeps even physically stolen data from being used. |
| Encryption at flight | Used to secure data during its move throughout a network. As data flows through a network, encrypting it keeps the data from being stolen. |
| Encryption in use | Encrypts data during usage to maintain secure data transfer. Using this method keeps in use data from being stolen. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is used to ensure that a user is who they say they are. This policy is used to secure each account for users. |
| Authorization | Authorization is used to check whether a user is allowed to access the specific content they are trying to access. |
| Accounting | Accounting is used to track what users do while on your program. You can account for a user’s footprint to make sure they are not performing any nefarious deeds. |

**\***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 |

**Resources**

safeonline. (2018, March 12). Secure coding practices: Digital security guide: Safeonline.ng : Digital Security Guide. Digital Security Guide | Safeonline.ng. <https://safeonline.ng/web-developers/secure-coding-practices/>

sei.cmu. (n.d.). SEI CERT C Coding Standard. SEI CERT C Coding Standard - SEI CERT C Coding Standard - Confluence. <https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>