**Green Pace Developer: Security Policy Guide**



# 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 | Properly validating input from CLI, network interfaces, environment variables, and user-controlled files can reduce many software vulnerabilities that arise from lack of validation.1 |
| 1. Heed Compiler Warnings | Compiler specific warnings are the highest warnings available for a specific compiler, and all warnings should be reduced by modifying code directly. This can be done through the use of static and dynamic tools created to eliminate security flaws.1 |
| 1. Architect and Design for Security Policies | Design software that enforces security policies. An example would be to utilize privileges through channels of intercommunicating subsets, each with their specific privileges set accordingly.1 |
| 1. Keep It Simple | It’s best to keep software designs simple, as complex designs increase the potential amount of errors that can be introduced into the software through implementation.1 |
| 1. Default Deny | Configure base access around permissions. The configured security can identify conditions where access is granted.1 |
| 1. Adhere to the Principle of Least Privilege | Each process should run with the minimal amount of privileges to do so. Elevated permissions should be granted for the smallest time window possible to allow a privileged operation to complete and nothing more.1 |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex systems (shells, databases, etc), should be sanitized to minimize risk of attackers gaining unauthorized access. SQL injection is an example that can be mitigated through data sanitization.1 |
| 1. Practice Defense in Depth | Implement multiple defensive strategies that allow attacks to be further defended and potentially eliminated in the event they pass any initial barriers.1 |
| 1. Use Effective Quality Assurance Techniques | Quality assurance can be impactful when identifying and eliminating vulnerabilities. Methods such as fuzz testing, pen testing, and source code audits should be incorporated as part of a QA program.1 |
| 1. Adopt a Secure Coding Standard | Create or implement secure coding standards for target development languages and platforms. This ensures consistency in the code and reduces inadvertent security holes from being introduced into a system.1 |

*1. Top 10 secure coding practices*. Top 10 Secure Coding Practices - CERT Secure Coding - Confluence. (n.d.). Retrieved January 20, 2023, from https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices

### 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** | **Obey the one-definition rule** |
| --- | --- | --- |
| **Data Type** | DCL60-CPP | Code intent can get confusing when multiple definitions exist. |

| **Noncompliant Code** |
| --- |
| This code demonstrates two different units defining a class with identical names, but different definitions. |
| // a.cpp  **struct** S {  **int** a;  };    // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| By using a header file that introduces the object into both translation units, the issue can be mitigated. |
| // S.h  **struct** S {  **int** a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** 4 - by keeping the design as simple as possible, side effects can be minimized |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-DCL60-a | A class, union, or enum name shall be a unique identifier |
| LDRA tool suite | 9.7.1 | 286 S, 287 S | Fully implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not rely on the value of a moved-from object** |
| --- | --- | --- |
| **Data Value** | EXP63-CPP | Values can change that may not seem obvious when moving objects. |

| **Noncompliant Code** |
| --- |
| Here we see integers from 0 to 9 expect to be printed to std.io from a std::string rvalue reference. Since the object was moved and reused assuming internal state had been cleared, the output can be unexpected. |
| #include <iostream>  #include <string>    **void** g(std::string v) {    std::cout << v << std::endl;  }    **void** f() {    std::string s;  **for** (unsigned i = 0; i < 10; ++i) {      s.append(1, **static\_cast**<**char**>('0' + i));      g(std::move(s));    }  } |

| **Compliant Code** |
| --- |
| The std::string is initialized to expected value throughout the loop iteration. This helps to maintain a that the object state is valid before accessing it. |
| #include <iostream>  #include <string>    **void** g(std::string v) {    std::cout << v << std::endl;  }    **void** f() {  **for** (unsigned i = 0; i < 10; ++i) {      std::string s(1, **static\_cast**<**char**>('0' + i));      g(std::move(s));    }  } |

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

| **Principles(s):** 3,7,9 – By preparing for worst case scenarios, eliminate assumptions that values won’t change |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-EXP63-a | Do not rely on the value of a moved-from object |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | The misuse of strings and “off by one errors” can lead to vulnerabilities such as buffer overflow. |

| **Noncompliant Code** |
| --- |
| Unbounded code that could lead to a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| This code constructs std::string read from an input stream that is at most 32 characters. |
| #include <fstream>  #include <string>    **void** f(std::istream &in) {  **char** buffer[32];  **try** {      in.read(buffer, **sizeof**(buffer));    } **catch** (std::ios\_base::failure &e) {      // Handle error    }    std::string str(buffer, in.gcount());    // ...  } |

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

| **Principles(s):** 1,8,9,10 – Off by one errors are quite common and having multiple security layers in place to adhere to minimizes holes from being left open |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| Polyspace Bug Finder | R2022b | CERT C++: STR50-CPP | Rule partially covered |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | STR02-C | Ensure invalid information doesn’t infiltrate a SQL database. |

| **Noncompliant Code** |
| --- |
| This code example inputs an email address to a buffer and uses a string argument to a system call. |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| By whitelisting acceptable characters, we can remove any characters that are not acceptable. |
| **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 = '\_';  } |

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

| **Principles(s):** 7 – This is fairly black and white from the principle, data sent to other systems should be sanatized |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Astree | 22.04 | N/A | Supported by stubbing/taint analysis |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Clear sensitive information stored in reusable resources** |
| --- | --- | --- |
| **Memory Protection** | MEM03-C | Ensure memory is being used effectively to prevent unwanted side effects. |

| **Noncompliant Code** |
| --- |
| Sensitive information is stored in dynamically allocated memory that is being referenced by secret, which eventually becomes processed and deallocated by calling free(). |
| **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** |
| --- |
| Here we see dynamic memory being sanitized before being freed. This is achieved by clearing allocated space. |
| **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; |

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

| **Principles(s):** 3,9 – By properly designing software and using effective QA, effective memory use can be addressed |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |
| CodeSonar | 7.2p0 | Customizable | Users can add custom check for use of realloc() |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Understand the termination behavior of assert() and abort()** |
| --- | --- | --- |
| **Assertions** | ERR06-C | Utilizing assert() and abort() properly can allow proper runtime assertions to be replaced and executed as intended. |

| **Noncompliant Code** |
| --- |
| This code shows a function that is called before the program exits to clean up. |
| **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** |
| --- |
| Here we see a call to assert() has an if statement that calls exit(). This ensures the necessary termination routines are executed. |
| **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):** 3,10 – A logic error can be present that causes side effects, design code accordingly |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PC-lint Plus | 1.4 | 586 | Full supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Guarantee exception safety** |
| --- | --- | --- |
| **Exceptions** | ERR56-CPP | Exceptions introduce code paths into programs and the potential effects of code taking these paths should be considered to minimize any unwanted effects that may occur. |

| **Noncompliant Code** |
| --- |
| This code shows a flawed copy assignment operator. If a new expression throws an exception, the function will modify state of both member variables. This can lead to undefined behavior. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }        IntArray(**const** IntArray& that); // nontrivial copy constructor    IntArray& operator=(**const** IntArray &rhs) {  **if** (**this** != &rhs) {  **delete**[] array;        array = nullptr;        nElems = rhs.nElems;  **if** (nElems) {          array = **new** **int**[nElems];          std::**memcpy**(array, rhs.array, nElems \* **sizeof**(\*array));        }      }  **return** \***this**;    }      // ...  }; |

| **Compliant Code** |
| --- |
| This code provides the strong exception safety guarantee. The function allocates new storage for the copy prior to changing the state of the object. The function also avoids the test for self-assignment. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }      IntArray(**const** IntArray& that); // nontrivial copy constructor      IntArray& operator=(**const** IntArray &rhs) {  **int** \*tmp = nullptr;  **if** (rhs.nElems) {        tmp = **new** **int**[rhs.nElems];        std::**memcpy**(tmp, rhs.array, rhs.nElems \* **sizeof**(\*array));      }  **delete**[] array;      array = tmp;      nElems = rhs.nElems;  **return** \***this**;    }      // ...  }; |

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

| **Principles(s):** 1,3,4,5,6,8,9 – Due to how exceptions can introduce code paths into programs, ensure best practices across all principles to minimize security holes from arising |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2022b | CERT C++: ERR56-CPP | Rule fully covered |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Never hard code sensitive information** |
| --- | --- | --- |
| Miscellaneous | MSC41-C | It is insecure to use literal strings that aren’t verified from user input. |

| **Noncompliant Code** |
| --- |
| This code passes an authentication string as a string literal. |
| /\* Returns nonzero if authenticated \*/  **int** authenticate(**const** **char**\* code);    **int** main() {  **if** (!authenticate("correct code")) {  **printf**("Authentication error\n");  **return** -1;    }    **printf**("Authentication successful\n");    // ...Work with system...  **return** 0;  } |

| **Compliant Code** |
| --- |
| This code requires the user to provide the authentication and erases it when finished. |
| /\* Returns nonzero if authenticated \*/  **int** authenticate(**const** **char**\* code);    **int** main() {  #define CODE\_LEN 50  **char** code[CODE\_LEN];  **printf**("Please enter your authentication code:\n");  **fgets**(code, **sizeof**(code), stdin);  **int** flag = authenticate(code);    memset\_s(code, **sizeof**(code), 0, **sizeof**(code));  **if** (!flag) {  **printf**("Access denied\n");  **return** -1;    }  **printf**("Access granted\n");    // ...Work with system...  **return** 0;  } |

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

| **Principles(s):** 1,3,7,8,9 – The use of passing strings that are hard coded can impact many domains, ensure strings aren’t hard coded to minimize threats |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2022b | CERT C: Rule MSC41-C | Rule partially covered (checks for hard coded sensitive data) |
| Parasoft C/C++ test | 2022.2 | CERT\_C-MSC41-a | Do not hard code string literals |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| Integer | INT30-C | Careless use of unsigned integers can lead to unexpected behavior and lead to vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code can lead to unexpected behavior due to unsigned integer wrapping because of result value used to allocate insufficient memory. |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This is a precondition test of the operands of addition that guarantees there isn’t a possibility of unsigned wrap occurring. |
| #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):** 1,3,8,9 – Adhering to validating input data and ensuring proper types are assigned as intended will result in secure code |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Integer-overflow | Fully checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input Output | FIO42-C | Improper handling of files and performing operations that operate on pointers that reference null objects can cause erratic program behavior. |

| **Noncompliant Code** |
| --- |
| Here a file is opened and not closed before the function returns. |
| #include <stdio.h>    **int** func(**const** **char** \*filename) {  **FILE** \*f = **fopen**(filename, "r");  **if** (NULL == f) {  **return** -1;    }    /\* ... \*/  **return** 0;  } |

| **Compliant Code** |
| --- |
| This code shows that the file pointed to by f is closed before it is returned to the caller. |
| #include <stdio.h>    **int** func(**const** **char** \*filename) {  **FILE** \*f = **fopen**(filename, "r");  **if** (NULL == f) {  **return** -1;    }    /\* ... \*/  **if** (**fclose**(f) == EOF) {  **return** -1;    }  **return** 0;  } |

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

| **Principles(s):** 1,3,6,8,9 – File operations should maintain clear methods of execution and handling that ensures closing of all open the moment they’re no longer required to minimize unwanted side effects |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 49 D | Partially implemented |
| Polyspace Bug Finder | R2022b | CERT C: Rule FIO41-C | Rule partially covered (checks for resource leak) |
| Astree | 22.04 | N/A | Supported, but no explicit checker |

### 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 has a well-established DevOps process and infrastructure, however, by merging these ideas with some additional details around implementing various tools, Green Pace will be equipped with a robust security framework to help them achieve their goals. Many of the automations surrounding the standards in this policy will reside in several areas, including assessing and planning, designing, building, maintenance, and monitoring. Some tools that will assist with identifying potential security holes from the highlighted standards in this policy are LDRA tool suite, Polyspace Bug Finder, Astree, Parasoft C/C++ test, PC lint plus, CodeSonar, and Coverity.

Through many of the stages in the DevSecOps cycle depicted above, the aforementioned tools will be of paramount importance to ensure the best, secure free code is established and maintained at Green Pace. Assessing and planning will capitalize from principle 3 of architecting and designing for security policies. Although no code is written at this stage in the cycle, code structure and interconnectedness along with logic flow can be discussed and depicted to observe if any security flaws could potentially be introduced. Where the automation tools make great impact is throughout the designing and building phases and well into maintenance and monitoring/detection stages. The tools that have been identified to assist with this critical part of the DevSecOps lifecycle will exist to catch any potential vulnerabilities that make it into pre-production as well as production code. By ensuring the proper tools mentioned earlier are setup throughout the design and building phases, unit tests as well as code for building can be marked to add an extra mechanism of security revision prior to production. Once production is live, the maintenance and future code revisions will also have secure coding tools to automate the tasks of spotting potentially harmful code from being pushed. This cycle will then have these added security layers as part of production, which will allow code corrections to be spotted actively throughout the lifetime of Green Pace’s applications, overall enhancing their defense in depth goals.

### 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 |
| STR02-C | High | Likely | Medium | P18 | L1 |
| MEM03-C | Medium | Unlikely | High | P2 | L3 |
| ERR06-C | Medium | Unlikely | Medium | P4 | L3 |
| INT30-C | High | Likely | High | P9 | L2 |
| MSC41-C | High | Probable | Medium | P12 | L1 |
| FIO42-C | Medium | Unlikely | Medium | P4 | L3 |
| ERR56-CPP | High | Likely | High | P9 | L2 |
| DCL60-CPP | High | Unlikely | High | P3 | L3 |
| EXP63-CPP | Medium | Probable | Medium | P8 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest refers to the notion of protecting data that is stored on disks, SSDs, and other forms of storage media.2 This is especially important when information is stored for user accounts and passwords in a database along with any sort of authorization keys. Several areas of sensitive information that can be protected from encryption span across user logins, database changes, user files access, etc. The use of this encryption strategy is to ensure sensitive data remains secure and this implementation of encryption should be used to add enhanced security. |
| Encryption at flight | Encryption at flight is the practice of encrypting information that is being transferred.3 This is usually achieved through the use of AES keys where senders and receivers are aware of the keys used to transmit the information, thereby blocking anyone outside the intended recipient from gaining access to the information.3 This policy should be applied to protect data that is being sent to Green Pace servers from client machines, both internally and externally. Such data transfers encompass adding new users to a database, user logins, and user file access. The inclusion of encryption at flight is another helpful tool to include in any good security policy and security cycle. |
| Encryption in use | Encryption in use is a hybrid approach that blends some of the ideas from both in rest and at flight encryption tactics, but ensures that data is never unsecured no matter the stage or location of the information.4 Encryption in use is almost mandatory as this will allow any sensitive data to never remain unencrypted and only accessible for those intended. |

2. Google. (n.d.). *Default encryption at rest documentation,* Google Cloud. Google. Retrieved February 12, 2023, from https://cloud.google.com/docs/security/encryption/default-encryption#:~:text=Encryption%20at%20rest%20is%20encryption,)%20algorithm%2C%20AES%2D256.

3. *Content encryption: In Flight and at rest*. IBM Aspera. (2020). Retrieved February 12, 2023, from https://www.ibm.com/docs/en/aspera-on-cloud?topic=encryption-content-in-flight-rest

4. Das, P. (2022, October 5). *In-use encryption – what it is and how companies benefit*. Sotero. Retrieved February 12, 2023, from https://www.soterosoft.com/blog/data-in-use-encryption-data-in-motion-encryption/#:~:text=In%2DUse%20encryption%20takes%20a,%2C%20cloud%2C%20or%20hybrid).

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the aspect of AAA that aims to prevent anyone other than an intended user from gaining access to information about their demographics and ensures their credentials align with who the user states they are.5 This applies to user logins where users must be verified and checked against parameters such as their location or IP address when logging in. Any changes to a database should only be verified and published against authentication, along with authorization.5 |
| Authorization | Authorization is tightly coupled with authentication, but differs in regards to privileges granted to specifics user groups and user roles.5 Authentication verifies who a user is, whereas authorization grants permissions to users once they’re authenticated. After a user logins to a system for example, they may have certain levels of access granted that allow them to add new users to a system, make modifications to a database, access certain files, or even allow/limit access permissions for other users.5 |
| Accounting | Accounting is concerned with maintaining a log of what a user does or what occurs regarding their activity while logged into an organization network or company system.5 Accounting can be used to determine a user’s trends, activity, and even raise suspicions if any of these look abnormal. At the same time, should a malicious user gain access to a system in which they aren’t authorized, accounting will maintain a record of what the attacker performed, especially if they don’t/can’t clean up these logging records. |

5.Fortinet. (n.d.). *What is AAA Security?* Retrieved February 11, 2023, from <https://www.fortinet.com/resources/cyberglossary/aaa-security>

**\***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 | 01/21/2023 | Draft/Security Policies Partially Complete | Eric Kleihege |  |
| 1.1 | 02/11/2023 | Revised Draft/Security Policies Complete | Eric Kleihege |  |
| [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 |