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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All external or untrusted input data must be validated thoroughly. This includes user data, but also things like files and photos. |
| 1. Heed Compiler Warnings | Address all compiler feedback including warnings. Use the highest level of warnings available for your language. |
| 1. Architect and Design for Security Policies | Design and architect your system with security policies in mind. This may mean user permissions, roles, and separating the system into modules. |
| 1. Keep It Simple | Keep the system design as simple as possible. Involving other systems adds complexity and possible security risk. |
| 1. Default Deny | No access is assumed for anyone to anything. All access must be specifically granted to those that require it. |
| 1. Adhere to the Principle of Least Privilege | Every job or application gets the least amount of privilege required for the task. If more is needed, it is granted and the removed as soon as that task is completed. This makes the systems less capable of being compromised and used against you in harmful ways. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data sent to other complex subsystems. This is necessary as this data could be intercepted and corrupted in transit to the other system. |
| 1. Practice Defense in Depth | Layer your security measures to provide greater security coverage. Each layer should address a different possible risk. This limits the ability for something to get through all the layers but also limits the affect if they do get through. As the systems should have less permission and be segmented from other systems. |
| 1. Use Effective Quality Assurance Techniques | Having a through Quality Assurance plan can be very effective at preventing issues. This should include multiple types of internal software testing. Another important piece is Code inspections during the development phase. External security audits can be useful in finding issues you did not anticipate as they may take a different approach. |
| 1. Adopt a Secure Coding Standard | Creating or adopting an existing secure coding standard will go a long way in preventing issues. This will remove many risks by providing a secure clear way to do specific vulnerable actions. This will also make code reviews and inspections more thorough as focus can be on other things. |

### 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** | **Minimize the scope of variables and functions** |
| --- | --- | --- |
| **Data Type** | STD-001-C | Variables and functions should be declared in the minimum scope from which all references to the identifier are still possible.  When a larger scope than necessary is used, code becomes less readable, harder to maintain, and more likely to reference unintended variables |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function counter() increments the global variable count and then returns immediately if this variable exceeds a maximum value |
| unsigned **int** count = 0;    **void** counter() {  **if** (count++ > MAX\_COUNT) **return**;    /\* ... \*/    } |

| **Compliant Code** |
| --- |
| In this compliant solution, the variable count is declared within the scope of the counter() function as a static variable. The static modifier, when applied to a local variable (one inside of a function), modifies the lifetime (duration) of the variable so that it persists for as long as the program does and does not disappear between invocations of the function. |
| **void** counter() {  **static** unsigned **int** count = 0;  **if** (count++ > MAX\_COUNT) **return**;    /\* ... \*/    } |

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

| **Principles(s):** 6. Least Privilege. Functions should only have access to the variables they need and no others. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Local object scope  Global object scope | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL19 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | [Properly encode relationships in constant definitions](https://wiki.sei.cmu.edu/confluence/display/c/DCL08-C.+Properly+encode+relationships+in+constant+definitions) |
| --- | --- | --- |
| **Data Value** | STD-002-C | If a relation exists between constants, you should encode the relationship in the definitions. Do not give two independent definitions, because a maintainer may fail to preserve that relationship when modifying the code. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the definition for OUT\_STR\_LEN must always be two greater than the definition of IN\_STR\_LEN. The following definitions fail to embody this relationship: |
| **enum** { IN\_STR\_LEN=18, OUT\_STR\_LEN=20 }; |

| **Compliant Code** |
| --- |
| The declaration in this compliant solution embodies the relationship between the two definitions: |
| **enum** { IN\_STR\_LEN=18, OUT\_STR\_LEN=IN\_STR\_LEN+2 }; |

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

| **Principles(s):** 2. Architect and design for Security Policy. The relationship between items should be coded not assumed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Local object scope  Global object scope | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | [Guarantee that storage for strings has sufficient space for character data and the null terminator](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Because the input is unbounded, the following code could lead to a buffer overflow. |

| **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;  } |

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

| **Principles(s):** 3. ValidateInput Data. Validate the data and save it safely in sufficient storage with room for null terminator. |
| --- |

**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 |
| CodeSonar | 7.2p0 | Misc.mem.nterm | No space for null terminator  Buffer overrun Type overrun |

#### Coding Standard 4

| **Coding Standard** | **Label** | [Prevent SQL injection](https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection) |
| --- | --- | --- |
| **SQL Injection** | STD-004-J | SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the [untrusted data](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-untrusteda) may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are [sanitization](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sa) and validation, which are typically implemented as parameterized queries and stored procedures. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a char array, the database connection is created, and then the passwords are hashed.  Unfortunately, this code example permits a SQL injection attack by incorporating the unsanitized input argument username into the SQL command, allowing an attacker to inject validuser' OR '1'='1. |
| **try** {        String pwd = hashPassword(password);        String sqlString = "select \* from db\_user where username=" +          username + " and password =" + pwd;        PreparedStatement stmt = connection.prepareStatement(sqlString);          ResultSet rs = stmt.executeQuery();  **if** (!rs.next()) {  **throw** **new** SecurityException("User name or password incorrect");        } |

| **Compliant Code** |
| --- |
| This compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long user name. |
| **try** {      String pwd = hashPassword(password);        // Validate username length  **if** (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString);      stmt.setString(1, username);      stmt.setString(2, pwd);      ResultSet rs = stmt.executeQuery();  **if** (!rs.next()) {  **throw** **new** SecurityException("User name or password incorrect");      } |

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

| **Principles(s):** 8. Defense in Depth and 1. Validate User Input. Preventing SQL injection is one solid defense in your plan. Validating the input before loading it can be another. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The CheckerFramework | 2.1.31 | Tainting Checker | Trust and security errors (see Chapter 8) |
| CodeSonar | 7.2p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |

#### Coding Standard 5

| **Coding Standard** | **Label** | [Free dynamically allocated memory when no longer needed](https://wiki.sei.cmu.edu/confluence/display/c/MEM31-C.+Free+dynamically+allocated+memory+when+no+longer+needed) |
| --- | --- | --- |
| **Memory Protection** | STD-005-C | Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to free() with that pointer value. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text\_buffer referring to the object. |
| #include <stdlib.h>    **enum** { BUFFER\_SIZE = 32 };    **int** f(**void**) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  **if** (text\_buffer == NULL) {  **return** -1;    }  **return** 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the pointer is deallocated with a call to free(). |
| #include <stdlib.h>    **enum** { BUFFER\_SIZE = 32 };    **int** f(**void**) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  **if** (text\_buffer == NULL) {  **return** -1;    }    **free**(text\_buffer);  **return** 0;  } |

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

| **Principles(s):** 4. Heed Compiler Warnings. Not freeing dynamically allocated memory can cause a lot of issues. Heeding all warnings will help issues like this to be fixed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Supported, but no explicit checker |
| Axivion Bauhaus Suite | 7.2.0 | CertC-Mem31 | Can detect dynamically allocated resources that are not freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | [Ensure that integer conversions do not result in lost or misinterpreted data](https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data) |
| --- | --- | --- |
| **Assertions** | STD-006-C | Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. This rule is particularly true for integer values that originate from untrusted sources and are used in any of the following ways. |

| **Noncompliant Code** |
| --- |
| Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code example results in a truncation error on most [implementations](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-implementation). |
| #include <limits.h>    **void** func(**void**) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    sc = (**signed** **char**)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type. |
| #include <limits.h>    **void** func(**void**) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;  **if** (u\_a <= SCHAR\_MAX) {      sc = (**signed** **char**)u\_a;  /\* Cast eliminates warning \*/    } **else** {      /\* Handle error \*/    }  } |

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

| **Principles(s):** 1. Validate data. Confirm the data is the type you believe it should be to prevent issues with casting. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| Code Sonar | 7.2p0 | LANG.CAST.PC.AV | Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer |

#### Coding Standard 7

| **Coding Standard** | **Label** | [Handle all exceptions thrown before main() begins executing](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR58-CPP.+Handle+all+exceptions+thrown+before+main%28%29+begins+executing) |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Not all exceptions can be caught, even with careful use of function-try-blocks |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the constructor for S may throw an exception that is not caught when globalS is constructed during program startup. |
| **struct** S {    S() noexcept(**false**);  };    **static** S globalS; |

| **Compliant Code** |
| --- |
| This compliant solution makes globalS into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for S will be executed the first time the function globalS() is called rather than at program startup. |
| **struct** S {    S() noexcept(**false**);  };    S &globalS() {  **try** {  **static** S s;  **return** s;    } **catch** (...) {      // Handle error, perhaps by logging it and gracefully terminating the application.    }    // Unreachable.  } |

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

| **Principles(s):** 9. Use Effective Quality Assurance. Confirming that all exceptions are handled before calling Main() is one way to perform quality assurance on your code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | **potentially-throwing-static-initialization** | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-ERR58** |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | [Use parentheses for precedence of operation](https://wiki.sei.cmu.edu/confluence/display/c/EXP00-C.+Use+parentheses+for+precedence+of+operation) |
| --- | --- | --- |
| Expressions | STD-008-C | C programmers commonly make errors regarding the precedence rules of C operators because of the unintuitive low-precedence levels of &, |, ^, <<, and >>. Mistakes regarding precedence rules can be avoided by the suitable use of parentheses. |

| **Noncompliant Code** |
| --- |
| The intent of the expression in this noncompliant code example is to test the least significant bit of x: |
| x & 1 == 0 |

| **Compliant Code** |
| --- |
| In this compliant solution, parentheses are used to ensure the expression evaluates as expected: |
| (x & 1) == 0 |

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

| **Principles(s):**2. Heed Compiler Warnings. Parenthesis is not a big issue, but could cause unexpected behavior. This a good example for heeding compiler warnings. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P | L |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Jarasoft Jtest | 2022.2 | **CERT.EXP53.APAREN** | Use '()' to separate complex expressions |
| SonarQube | 6.7 | S864 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | [Do not read uninitialized memory](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory) |
| --- | --- | --- |
| Expressions | STD-009-CPP | Local, automatic variables assume unexpected values if they are read before they are initialized. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |
| #include <iostream>    **void** f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>    **void** f() {  **int** i = 0;    std::cout << i;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard. This is a code example of a standard that a team could adopt. Initializing memory is a solid coding practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | **uninitialized-local-read** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **uninitialized-variable-use** |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | [Close files when they are no longer needed](https://wiki.sei.cmu.edu/confluence/display/c/FIO42-C.+Close+files+when+they+are+no+longer+needed) |
| --- | --- | --- |
| Input Output | STD-010-C | A call to the fopen() or freopen() function must be matched with a call to fclose() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first. |

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

| **Compliant Code** |
| --- |
| In this compliant solution, the file pointed to by f is closed before returning 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):** 8. Practice Defense in Depth. Not only do you need to worry about the pointer but you want to not have the file open any longer than required. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | [Insert text.] | Supported, but no explicit checke |
| CodeSonar | 7.2p0 | ALLOC.LEAK | Leak |

### 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.

Most of the security standards in this document affect several pre-production phases. Specifically, the Design, Build, and Verify and Test phases. For the Design phase one standard that comes to mind is Standard 1. To minimize the scope of variables and functions you need to create a plan in the design phase. Laying out your functions and variables will guarantee a clean scope for both. Standard 10 would also require some thought in the design phase to plan when to open a file, do what needs to happen and then close it as soon as possible. The Build and Verify and test phases are where you could use a build tool like Jenkins to run all your automated testing automatically after every build. This can be set up to run every time, so it is automatic and consistent. The Monitor and Detect phase are where you could analyze logs for SQL Injection attempts.

### 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-C | Low | Unlikely | Medium | P2 | L3 |
| STD-002-C | Low | Unlikely | High | P1 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-J | High | Probable | Medium | P12 | L1 |
| STD-005-C | Medium | Probable | Medium | P8 | L2 |
| STD-006-C | High | Probable | High | P6 | L2 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-C | Low | Likely | Low | P | L |
| STD-009-CPP | High | Probable | Medium | P12 | L1 |
| STD-010-C | Medium | Unlikely | Medium | P4 | L3 |

### 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 | Encrypted at rest is encrypted data not currently being moved. This data could be on a hard drive, server, usb drive etc. This data is often more valuable because there is a higher quantity of sensitive data. Measures must be taken to assure encrypted data at rest is physically secured. One of the first steps is identifying the data and classifying it. |
| Encryption at flight | Encryption in flight is encrypted data currently being transferred from one system to another, through the cloud, from one network to another, etc. This data is often valuable data like usernames and passwords. Having the data encrypted so it can be replaced but not unencrypted increases your security. As does moving the data on encrypted systems like https, ssl, tls. |
| Encryption in use | Encryption in use is encrypted data currently at use by a program. When a program is using, editing, modifying, or deleting encrypted data it can be at risk. One step in protecting this data is to restrict access. Another it to keep the data encrypted during use. System visibility will also help you know if there is an issue. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of authenticating you are who you say you are. This can be done with a username and password. It can also include additional authentication measures like MFA, single sign on, biometrics, digital certificates, and public key infrastructure. This policy can apply to a user gaining access to any application, system, server, or SaaS. |
| Authorization | Authorization is usually applied after authentication is complete. Authorization is the process of allowing the authenticated user access to certain areas and commands and denying access to others. The level of access allowed can be based on many things like physical location, frequency of use, time of day, ip address, etc. |
| Accounting | Accounting is the practice of logging the authenticated and authorized user’s actions, length of time in the system, data received and sent, etc. This data can then be reviewed to see which users tried to access which systems, where access was granted, and what was done with the access. The data can show files accessed, databases modified, users added and deleted, etc. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.5 | 01/21/2023 | Added security principles and coding standards | Amy Vredevoogd | Professor |
| 2.0 | 02/14/2023 | Completed Document | Amy Vredevoogd |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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