**Green Pace Developer: Security Policy Guide – Ryan St George**



# 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.

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All inputted data ensures that data from untrusted sources should be verified and ensure there is no malicious code before being processed. This can prevent a wide array of issues including SQL Injection and XSS. |
| 1. Heed Compiler Warnings | Compiler warnings of potential security flaws or bugs in the code. By keeping an eye and eliminating all these warnings are a preemptive measure to rectify any security vulnerabilities. |
| 1. Architect and Design for Security Policies | This involves creating systems with security in mind from the beginning. Security should be an integral part of the architecture and design phases and not be an afterthought. |
| 1. Keep It Simple | A simple design is easier to test and verify any vulnerabilities. Having unnecessary features allows for more possible security threats. Keep the security, and the application simple. |
| 1. Default Deny | Simply do not give permissions unless they are granted. For instance, if a type of access is not allowed, it should not be given by default. Minimizing permissions also minimizes the attack surface. |
| 1. Adhere to the Principle of Least Privilege | Similarly, to Default Deny, only grant what is required to complete a task/job. Giving too much privilege that is not required can cause more harm than good. Limiting the damage that can occur. Also review permissions often to revoke unnecessary permissions. |
| 1. Sanitize Data Sent to Other Systems | When transferring data to external systems it should be sanitized to prevent any propagation of security issues. Ensuring the data can’t be used to carry out attacks such as SQL injection. You should strip any unnecessary content before sending it out. |
| 1. Practice Defense in Depth | Having a layered approach for security that involves multiple defensive strategies in case one fails. From firewalls, IDS’s, antivirus software, and many other tactics. |
| 1. Use Effective Quality Assurance Techniques | Having QA allows for identification and addressing any security issues before the software is deployed. This can include code reviews, automated testing, pen testing, and even manual testing to ensure the code is secure against any attacks. Making the software that much more resistant. |
| 1. Adopt a Secure Coding Standard | By having secure coding standards which are a set of rules and guidelines to prevent security vulnerabilities. By having developers adhere to these standards allow for a baseline of standards in their code and other tasks. Oftentimes updated frequently to include new standards from newly discovered threats. |

### 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-CPP] | Enforce Strict Data Typing |

| **Noncompliant Code** |
| --- |
| As shown using generic types allows for potential unpredictable behaviors that are often overlooked. Sometimes even causing type errors during runtime. |
| Object variable = “Hello, World”; |

| **Compliant Code** |
| --- |
| In this example we can see we are more specific with the type to ensure the safety and prevent any type errors at runtime. Overall you want to ensure you are picking the best data type for the variable. |
| String variable = “Hello, World”; // Now a strict data type |

| **Principles(s):** 1. Least privilege which allows for the least amount of privilege required in terms of the type access. This ensures the type cannot be changed maliciously or by accident. 2. Keep it simple which allows for strict data typing simplifies the code understanding and readability. Having complex code that isn’t clear oftentimes causes confusion and security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analyzer | 1.4.3 | TypeSafetyCheck | Tool Analyzes code to ensure that all variables are declared with the most specific and appropriate data type available. |
| CppCheck | 2.12.1 | UnsafeTypeUsage | Scans code for generic type declarations and recommends type specific alternatives. |
| Clang-Tidy | 18.0.0 | Modernize use auto | Can flag the use of explicit type declarations that could be replaced with ‘auto’ to avoid mismatches. |
| SonarQube | 10.3.0 | S1001 | Can inspect code for type related practices and compliance with current standards. |

#### 

#### 

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Preventing Data Value Range Errors |

| **Noncompliant Code** |
| --- |
| Truncation can occur when a value is too small to represent the result, and conversions can result in values out of range in the resulting type. |
| Unsigned long int ul = ULONG\_MAX;  Signed char sc;  sc = (signed char)ul; // Implicit cast may cause overflow |

| **Compliant Code** |
| --- |
| In this scenario you can see that before casting we’ll check if the value is within the range of the target type to prevent the data loss on overflow errors. Note this specific error we did not complete the else statement as we are just trying to get the point across. |
| Unsigned long int ul = ULONG\_MAX;  Signed char sc;  If (ul <= static\_cast<unsigned long>(SCHAR\_MAX)) {  sc = static\_cast<signed char>(ul);  } else {  // Add the custom compliant error handling  } |

| **Principles(s):** 1. Validate the input data to ensure that all inputs are properly validated to be within the expected ranges to prevent these errors.  2. Architect and Design for Security policies to ensure the compliance code includes provisions for the handling out of range data showing that the security has been integrated into the design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.9.0 | CONVERSION\_ERROR | Can identify potential data conversion errors where truncation or sign extension could occur |
| CppCheck | 2.12.1 | dangerousCast | Can warn about casts that could result in losing data, such as from a larger to smaller data type. |
| Clang Static Analyzer | 18.0.0 | CastRangeChecker | Analyzes casts to ensure their values are within the range of the target type. |
| SonarQube | 10.3.0 | S5632 | Can detect when a cast could lead to data loss due to a change in the data types range. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensuring correct string sizes and not neglecting buffer boundaries. This can lead to overflows and runtime errors. |

| **Noncompliant Code** |
| --- |
| Here if a user inputs more than the buffer allows it will result in a buffer overflow. |
| #include <iostream>  int main(void) {  char buffer[10];  std::cin >> buffer;  std::cout << “text: “ << buffer << ‘\n’;  } |

| **Compliant Code** |
| --- |
| We eliminated the buffer overflow by using width to limit the field width by the size of the buffer set. |
| #include <iostream>  int main(void) {  char buffer[10];  std::cin.width(12);  std:cin >> buffer;  std::cout << “text: “ << buffer << ‘\n’;  } |

| **Principles(s):** 1. Data validation which is about ensuring that all data inputs are checked to prevent errors and vulnerabilities. 2. Default Deny which ensures unless explicitly allowed it should be denied preventing possible overflows and vulnerabilities. 3. Adopt a secure coding standard which implies using safer methods for handling strings to prevent buffer overflows. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.12.1 | bufferAccessOutOfBounds | Detects cases where buffer accesses that may be out of bounds. |
| Clang Static Analyzer | 18.0.0 | ArrayBoundV2 | Can check for potential buffer overflows in array or pointers. |
| Coverity | 2023.9.0 | OVERRUN | Identifies instances where untrusted input may overflow a buffer. |
| Fortify | 22.1.0 | Buffer Overflow | Can scan for potential buffer overflows by checking the size of data types and memory allocation. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Limit inputs in text fields, inputs, and network connections. Limit the string concatenation as this is the primary point of entry for script injection and use parameterized queries in SQL. |

| **Noncompliant Code** |
| --- |
| This code concatenates user input into an SQL query. Clearly allowing for SQL injection if the user inputs contain commands. |
| std::string user\_input = "'; DROP TABLE users; --";  std::string query = "SELECT \* FROM users WHERE name = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| This code uses parameterized query which ensures the user input is treated as a parameter and not part of the SQL command. This prevents the injection. |
| std::string user\_input = getUserInput();  SQLQuery query = SQLQuery("SELECT \* FROM users WHERE name = ?");  query.addParameter(user\_input); |

| **Principles(s):** 1. Input validation which ensures all input is verified before use, particularly in constructing SQL queries. 2. Adopt a secure coding standard ensuring that all database queries are parameterized which is essential to protect against SQL injection. 3. Defense in Depth which uses multiple layers of security to protect against SQL injection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Medium | Highest | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.14.0 | SQLInjectionChecker | Open-source security scanner that can detect SQL injection. |
| Fortify | 22.1.0 | SQL\_Injection | Static code analyzer that can scan for patterns that may lead to a SQL injection. |
| SonarQube | 10.3.0 | Sql-injection | Scans source code for potential SQL injection and highlights vulnerable SQL query instructions. |
| Checkmarx | 9.3.0 | SqlInjection | Provides static code analysis to identify SQL injection vulnerabilities across multiple coding languages. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Preventing access to memory that is not granted. For instance someone should not be granted access to freed memory, read uninitialized memory, and you should detect and handle these errors as they come. Preventing the stack buffer overflow. |

| **Noncompliant Code** |
| --- |
| As shown here in this code it takes the memory, uses it, then frees it. This returns the pointer to the freed memory which can lead to some bad behaviors if dereferenced. |
| char \*allocateAndUse() {  char \*buffer = new char[10];  // simulated use of the buffer  delete [] buffer;  return buffer;  } |

| **Compliant Code** |
| --- |
| Using std::unique\_ptr for the memory allocation ensures the memory is automatically freed when the pointer goes out of the scope. This prevents the access of the freed memory. This also returns a smart pointer allowing for a safe transfer of ownership of the memory. |
| std::unique\_ptr<char[]> allocateAndUse() {  std::unique\_ptr<char[]> buffer = std::make\_unique<char[]>(10);  // simulated use of the buffer  return buffer; // Returned much safer  } |

| **Principles(s):** 1. Keep it simple with simplified code that can automatically handle deallocation, which reduces the complexity. 2. Adhere to the principle of least privilege which ensures that the caller only has access to the memory if it has proper ownership. 3. Use effective QA techniques such as smart pointers to help prevent common memory management errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.22.0 | Memcheck | Monitors memory management in real time to detect memory leaks. |
| AddressSanitizer | LLVM 3.1+ | UseAfterFree | Fast memory error detector that can identify use after free bugs. |
| Cppcheck | 2.12.1 | invalidPointer | Static analysis tool that checks for invalid pointer use including after it was freed. |
| Coverity | 2023.9.0 | USE\_AFTER\_FREE | Identifies instances where objects are being used after their memory was freed. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to handle errors that test the conditions that should never be true. This of course only works if the code is correct. The purpose is for the developer to see the assertion. Often used in testing for error conditions within the program. |

| **Noncompliant Code** |
| --- |
| Here we can see the code divides without checking to see if the denominator is 0. If it is 0 the program may crash due to an undefined result. |
| int divide(int numerator, int denominator) {  return numerator / denominator;  } |

| **Compliant Code** |
| --- |
| In this case before performing the division we use an assertion to verify the denominator is not 0. If it is, it will abort alerting the programmer of the error. Also if you have assertions, ensure they are checked up on and not set to the public and forgotten about. This is why oftentimes in advanced code its good to have code for the failsafe. |
| #include <cassert>  int divide(int numerator, int denominator) {  assert(denominator != 0); // Assert that the denominator is not zero  return numerator / denominator;  } |

| **Principles(s):** 1. Keep it simple where assertions keep the program logic simple by clearly stating the assumptions made at specific points. 2. Adopt a secure coding standard that includes using assertions during development to validate assumptions. 3. Use effective QA techniques to catch logical errors early on and prior to deployment/release. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analysis Tool | 1.4.3 | AssertUsage | Checks for proper use of assertions in the code and that they are not disabled in production builds. |
| Cppcheck | 2.12.1 | AssertMisuse | Can detect misuse of assertions such as using them to control the flow instead of just for debugging. |
| Clang Static Analyzer | 18.0.0 | AssertSideEffects | Ensures that assertions do not contain side effects, which should remain true regardless. |
| Coverity | 2023.9.0 | ASSERT\_SIDE\_EFFECT | Can identify assertions that might have side effects, potentially affecting the programs state. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | An exception should be thrown when a situation has happened that prevents the correct execution of a function from being completed. |

| **Noncompliant Code** |
| --- |
| In this function an error message is printed and -1 is returned. This mixes the error reporting and normal return values of the calculation and does not enforce the handling of the error. |
| int calculateDivision(int numerator, int denominator) {  if (denominator == 0) {  std::cerr << "Error: Division by zero is undefined." << std::endl;  return -1;  }  return numerator / denominator;  } |

| **Compliant Code** |
| --- |
| This function using the invalid\_argument throws the exception when a division by 0 is attempted. This enforces error handling and separates this from a normal return value. This can get much more complex especially in using arrays. |
| #include <stdexcept>  int calculateDivision(int numerator, int denominator) {  if (denominator == 0) {  throw std::invalid\_argument("Division by 0 is not allowed.");  }  return numerator / denominator;  } |

| **Principles(s):** 1. Error handling and prevention by using exceptions we can enforce proper error handling ensuring that an error is managed in a controlled way such as division by zero. 2. Keep it simple by throwing an exception and clearly indicating the problem. 3. Adopt a secure coding standard to ensure exceptions are a part of these standards so they are properly handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analysis Tool | 1.4.3 | ExceptionSafety | Verifies the functions are using exceptions correctly for error handling and properly documented. |
| Cppcheck | 2.12.1 | divisionByZero | Detects divide by zero errors and recommends using exceptions. |
| Clang Static Analyzer | 18.0.0 | DivisionByZero | Can analyze for division by zero and suggests improvements. |
| Coverity | 2023.9.0 | DIVIDE\_BY\_ZERO | Identifies divide by zero errors and checks if they are properly handled. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Naming Conventions | [STD-008-CPP] | Ensure good naming conventions as poor ones can create confusion and errors. |

| **Noncompliant Code** |
| --- |
| Having poor naming conventions its oftentimes hard for someone else to look at this and understand what they are. Or even when a program gets so big you oftentimes forget. Keep it simple, yet proper. |
| int a;  double li; |

| **Compliant Code** |
| --- |
| Having proper, oftentimes bigger names is totally ok. Also don’t forget to stay in standards and use camelCase or underscores. Without them its harder to read, and of course outside standards. |
| int accountBalance;  double loanInterest; |

| **Principles(s):** 1. Keep it simple with good naming conventions to make the code clear and easy to understand. 2. Adopt a secure coding standard to ensure standard naming variables and functions. This helps in maintaining the code especially if multiple people touch it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3.0 | CodeSmell | Detects nonstandard naming conventions and suggests changes. |
| StyleCop | 6.2.0 | SA1306 | Can enforce the use of camelCase for variable names and checks for compliance. |
| Pylint | 3.0.2 | Variable-name | Checks python code to the naming standards which is configurable. |
| ESLint | 8.54.0 | Id-match | Enforces naming conventions in JavaScript code and is also configurable. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Unit Testing Conventions | [STD-009-CPP] | Unit Testing is great, but you must comply with standards. Having proper naming conventions, tests that have different behaviors, and cover a high percentage of code coverage. |

| **Noncompliant Code** |
| --- |
| By having poor test names, multiple assertions in a single test also is not good or correct. |
| void test1() {  Calculator calc;  assert(calc.add(1, 1) == 2);  assert(calc.subtract(2, 1) == 1); // Bad behavior  }  void test2() {  Calculator calc;  assert(calc.multiply(3, 5) == 15);  } |

| **Compliant Code** |
| --- |
| Here is a proper set of tests. You can see single assertions in a test, along with great naming conventions. This will help in creating the tests but also when the tests fail what they are trying to test. |
| void testAddition\_TwoPositiveNumbers\_ReturnsCorrectSum() {  Calculator calc;  assert(calc.add(1, 1) == 2); // Single behavior  }  void testSubtraction\_TwoPositiveNumbers\_ReturnsCorrectDifference() {  Calculator calc;  assert(calc.subtract(2, 1) == 1);  }  void testMultiplication\_TwoPositiveNumbers\_ReturnsCorrectProduct() {  Calculator calc;  assert(calc.multiply(3, 5) == 15);  } |

| **Principles(s):** 1. Keep it simple with single assertions in a test and clear naming conventions. 2. Use effective QA techniques with clear intents to catch bugs early. 3. Adopt Secure Coding Standards with consistent and comprehensive unit testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| JUnit | 5.10.1 | NamingConventionsChecker | Ensures that test methods in Java follow the proper naming conventions. |
| NUnit | 3.13.3 | SingleAssertPerTest | Analyzes tests to ensure each test contains a single assertion. |
| PyTest | 7.4.3 | TestNamingConvention | Ensures test functions in python are named correctly and follow best practices. |
| ESLint-plugin-testing-library | 6.1.0 | ConsistentTestItNaming | Enforce naming conventions for test blocks and assertions in JS. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-010-CPP] | Ensuring expressions do not read uninitialized memory. |

| **Noncompliant Code** |
| --- |
| Here you can see the reading of the uninitialized variable. This is not proper standard and shouldn’t be used to begin with. |
| int a;  bool condition = true;  if (condition) {  a = 10;  }  int result = a + 5; // a may be uninitialized if condition is false |

| **Compliant Code** |
| --- |
| Here we can see no matter the path of execution a will have a valid value when used in this expression. This is oftentimes why its important to initialize at the point of declaration as shown below. |
| int a = 0; // Initialize with a default value  bool condition = true;  if (condition) {  a = 10;  }  int result = a + 5; // a is guaranteed to be initialized |

| **Principles(s):** 1. Input validation to ensure all variables are initialized before use. 2. Keep it Simple to simplify how a variable should be initialized. 3. Use Effective Quality Assurance (QA) Techniques to test and ensure the code behaves properly under all conditions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.22.0 | UninitValue | Can detect use of uninitialized memory. |
| Cppcheck | 2.12.1 | uninitvar | Static code analysis tool that checks for uninitialized variables. |
| Clang Static Analyzer | 18.0.0 | UninitializedVariable | Analyzes code to find variables that may be used uninitialized. |
| Coverity | 2023.9.0 | UNINIT | Can identify use of uninitialized variables in the code. |

### 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.

Automation can be integrated during the build process by transforming tasks into automated ones within a CI/CD pipeline. Thus, enhancing team collaboration. This can involve utilizing tools like Docker for managing containerized applications, and even employing GitLab for source control management.

In the realm of SecOps automation can streamline the deployment of virtualized containers. The incorporation of automated security assessments and regression testing is critical during the quality assurance phase.

For monitoring and detecting it is beneficial to incorporate static application security testing into routine nightly builds. While focusing on critical portions of the codebase. Additionally integrating dynamic application security testing into the SDLC can provide an advantage of identifying vulnerabilities in a live environment. Even using tools like OWASP Dependency-Check for scrutinizing the security of code dependencies and aiding in the early detection of possible vulnerabilities.

### 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 | Likely | High | High | L2 |
| STD-002-CPP | Medium | Moderate | Medium | High | L3 |
| STD-003-CPP | High | High | Low | High | L1 |
| STD-004-CPP | Critical | High | Medium | Highest | L1 |
| STD-005-CPP | High | Medium | Medium | High | L2 |
| STD-006-CPP | Medium | Low | Low | Medium | L3 |
| STD-007-CPP | Medium | Low | Low | Medium | L3 |
| STD-008-CPP | Low | High | Low | Medium | L3 |
| STD-009-CPP | Low | Medium | Low | Medium | L3 |
| STD-010-CPP | Medium | High | Low | High | 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 at rest is aimed at securing the inactive data stored on any digital media. This is crucial because it ensures it’s not easily accessible or readable if the storage medium is lost, stolen, or compromised. The policy applies to safeguard the sensitive information against threats. |
| Encryption at flight | Often referred as encryption in transit which protects data as it travels across a network. This is essential for maintaining confidentiality and integrity of data as it moves. This policy is enforced to prevent interception, eavesdropping, and MiM attacks. |
| Encryption in use | Encryption in use protects data that is being processed by applications and systems. Ensures that data remains encrypted not only when its stored or moved, but also when its being utilized by computing processes. This policy is applicable to prevent unauthorized access or modification of sensitive data while it is being processed. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication which is the process of identifying a user or device through verification. Usually through verifying credentials or tokens. The policy applies as it’s the first step in ensuring access before any other interaction in an application or system. User logins are typically verified through multi factor authentication and complex passwords. |
| Authorization | Authorization occurs after authentication and involves granting or denying access/privileges. Allowing the user access to only the resources they are allowed. This policy applies to enforce restrictions on authenticated users and what they can do. For instance, making changes to a database, add new users, and overall user level access. You should review this often to ensure people don’t have access for longer than they need or even if they still exist within the company. |
| Accounting | Accounting often referred to as auditing is the process of logging user actions and activities. This can be from password attempts, logins, and overall modifications in a database. Having traceable records is important for security. Especially in detecting any vulnerabilities and threats. Allowing to see what files were accessed is really important. |

### Map the Principles

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

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

**Note not all are described in Coding Standards.**

Data Type

Principles: 1, 4, 10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_1)

Data Value

Principles: 1,5,8,10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_2)

String Correctness

Principles: 1,4,7,10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_3)

SQL Injection

Principles: 1,3,7,8,10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_4)

Memory Protection

Principles: 4, 6, 8, 9, 10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_5)

Assertions

Principles: 4, 9

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_6)

Exceptions

Principles: 3, 4, 9, 10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_7)

Naming Conventions

Principles: 4, 9, 10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_8)

Unit Testing Conventions

Principles: 4, 9, 10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_9)

Expressions

Principles: 1, 4, 10

Explanation: [CLICK HERE TO JUMP TO REASON](#_Coding_Standard_10)

## 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 |  |
| 1.1 | 11/09/2023 | Adding 10 principles, and 10 standards | Ryan St George | Professor Gray |
| 1.2 | 11/29/2023 | Completed Document | Ryan St George | Professor Gray |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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