**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 | Any and all external input into a program should be treated as untrustworthy. [FIO30-C](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings) is a reminder that all user input needs to be sanitized. Proper sanitization and validation of external data verifies that data meets format, range, and type requirements. |
| 1. Heed Compiler Warnings | Compiler warnings act as an extra layer of defense that indicates bugs and unsafe behavior. These issues can lead to vulnerabilities that may be exploited by attackers. [MSC00-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC00-C.+Compile+cleanly+at+high+warning+levels) recommends configuring compilers to the highest warning level and resolving all warnings before compiling. |
| 1. Architect and Design for Security Policies | Security needs to be at the forefront during development. Developing clean, correct code from the early stages leads to fewer vulnerabilities later. [MSC12-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC12-C.+Detect+and+remove+code+that+has+no+effect+or+is+never+executed), [MSC13-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC13-C.+Detect+and+remove+unused+values), [MSC14-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC14-C.+Do+not+introduce+unnecessary+platform+dependencies), and [MSC15-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC15-C.+Do+not+depend+on+undefined+behavior) are prime examples of this philosophy, by recommending to remove any and all dead code, unnecessary dependencies, and undefined behavior that arises on execution. |
| 1. Keep It Simple | Complicated lines of code are difficult to troubleshoot, leading to the possibility of them being exploited for unintended behaviors. Generic, frequently reused variables are also difficult to diagnose and lead to confusing code. [DLC01-C](https://wiki.sei.cmu.edu/confluence/display/c/DCL01-C.+Do+not+reuse+variable+names+in+subscopes) recommends avoidance with reusing variables. Always create a new variable that is clearly defined and visually distinct, as recommended in [DLC02-C](https://wiki.sei.cmu.edu/confluence/display/c/DCL02-C.+Use+visually+distinct+identifiers). |
| 1. Default Deny | By default, all access and authorization should be denied unless explicitly allowed and verified. Do not allow access to the system, as recommended in [ENV33-C](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177); never call the system() function, as it is open to vulnerabilities that can allow unauthorized access to the system. |
| 1. Adhere to the Principle of Least Privilege | Code should never have more access than is required to perform its purpose. If code becomes compromised, the limitation of its privileges contribute to containing the potential damage. |
| 1. Sanitize Data Sent to Other Systems | Any data that is used for other system components or APIs need to be sanitized and verified for correctness in order to mitigate malicious injection attempts or faulty program behavior. Even known-good files and location should be sanitized and verified, following the recommendation outlined in [FIO03-C](https://wiki.sei.cmu.edu/confluence/display/c/FIO03-C.+Do+not+make+assumptions+about+fopen%28%29+and+file+creation). |
| 1. Practice Defense in Depth | Single points of failure should be avoided. Multiple layers of security techniques should be employed at every level, so that in the event one security mechanism fails, there are others in place to mitigate the impact of vulnerabilities or exploits. Examples include a combination of techniques such as input validation, bounds checking, and proper access controls to protect against user input. |
| 1. Use Effective Quality Assurance Techniques | Testing techniques should always be utilized during the development process to save time and prevent excessive debugging later. Performing code reviews, heeding compiler warnings, properly documenting code, and performing unit tests and static analysis are important, as recommended in [MSC00-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC00-C.+Compile+cleanly+at+high+warning+levels) and [MSC04-C](https://wiki.sei.cmu.edu/confluence/display/c/MSC04-C.+Use+comments+consistently+and+in+a+readable+fashion). |
| 1. Adopt a Secure Coding Standard | All developers should maintain up-to-date education on secure coding practices, such as adhering to the outlined [SEI CERT C Coding Standards](https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard). Writing maintainable, encapsulated, clean code should be the primary goal of developers. |

### 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 non-compliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Avoid Implicit Type Conversions |

| **Noncompliant Code** |
| --- |
| The following code mixes variables of differing types: a signed int and an unsigned int. This can lead to difficult to diagnose logic errors on different architectures, where the inequality may resolve to false on x64 and true on x86.  Source: [https://blog.regehr.org/archives/268](https://blog.regehr.org/archives/268%20) |
| int a = -1;  unsigned int b = 1;  if (a > b) {  std::cout << “a is greater than b” << std::endl;  } |

| **Compliant Code** |
| --- |
| The following code avoids mixing signed and unsigned integers to prevent any unintended behavior. |
| int a = -1;  int b = 1;  if (a > b) {  std::cout << “a is greater than b” << std::endl;  } |

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

| **Principles(s):**   * Principle of Least Astonishment – Ensure code behaves in a predictable manner. Implicit type conversions can lead to unexpected results. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Hicpp-signed-bitwise | Detects uses of bitwise operators on signed integers. |
| cppcheck | 2.18 | Sign conversion | Identifies conversion problems of signed/unsigned integers at compile |
| SonarQube | 25.8 | Cpp:S3599 | Flags implicit type conversions. |
| PVS-Studio | 7.26 | V629, V642 | Flags comparisons between signed and unsigned values. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validate Input Ranges |

| **Noncompliant Code** |
| --- |
| The following code requests input from a user about their age. The code does not verify or sanitize the input using any intuitive logic. |
| int age;  std::cin >> age;  std::cout << “Age: “ << age << std::endl; |

| **Compliant Code** |
| --- |
| The following code requests user input about their age, but properly checks the stored input range to ensure it fits the expected range of a traditional human’s life expectancy. If the age is unexpected, the program terminates with an error message. |
| unsigned int age;  std:: cin >> age;  if (age < 0 || age > 120) {  std::cout << “Invalid age.” << std::endl;  return 0;  }  std::cout << “Age: “ << age << std::endl; |

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

| **Principles(s):**   * Principle of Fail-Safe Defaults – Inputs should always be considered unsafe. Validating user input to required specifications and reasonable ranges, the program will avoid unpredictable behavior, crashes, and potential security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Cert-str34-c | Ensures inputs are validated when used. |
| cppcheck | 2.18 | invalidscanf | Flags cases where user input is read incorrectly, or is out of bounds. |
| SonarQube | 25.8 | Cpp:S3649 | Flags user input that is not validated before being used. |
| Coverity | 2024.12 | Tainted\_scalar | Flags user input used without validation. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Avoid Dangerous String Functions |

| **Noncompliant Code** |
| --- |
| The following code segment utilizes the strcpy() function from the cstring library. It copies a name into a fixed-length string of characters without checking if it fits, which strcpy() does not have the built in ability to do. This can lead to buffer overflow vulnerabilities that can easily be exploited. |
| #include <cstring>  char name[10];  strcpy(name, “This string is too long.”);  std::cout << “Name: “ << name << std::endl; |

| **Compliant Code** |
| --- |
| This version of the above code opts for the use of the string variable instead, which removes the size limit and handles memory safely, which strcpy() does not. |
| #include <string>  std::string name = “This name is too long.”;  std::cout << “Name: “ << name << std::endl; |

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

| **Principles(s):**   * Principle of Least Privilege – Only give systems and users enough permissions to accomplish their tasks. strcpy() does not have built in checking tools, which assigns too much power to a function. Std::string avoids this. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Cppcoreguidelines-pro-bounds-array-to-pointer-decay | Flags all array to pointer decays. |
| cppcheck | 2.18 | dangerous function | Flags use of dangerous string functions like strcpy() |
| SonarQube | 25.8 | Cpp:S1075 | Flags unsafe string functions that may cause buffer overflows. |
| Coverity | 2024.12 | Buffer\_size | Flags potential buffer overflows from unsafe string handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use Prepared Statements to Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| The following code segment places input gathered from a user directly into the SQL query. This can lead to injection attack exploits, such as ‘or 1=1;’. |
| std::string query = “SELECT \* FROM users WHERE name = ‘” + name + “’;”;  sqlite3\_exec(db, query.c\_str(), nullptr, nullptr, nullptr); |

| **Compliant Code** |
| --- |
| The revised code using a prepared statement forces the query to remain static and counters SQL injection attempts by using a pre-built statement that cannot be tampered with. |
| sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, "SELECT \* FROM users WHERE name = ?", -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, name.c\_str(), -1, SQLITE\_STATIC);  sqlite3\_step(stmt);  sqlite3\_finalize(stmt); |

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

| **Principles(s):**   * Principle of Complete Mediation – Ensures that every request to an object or resource is checked and validated, every single time. Prepared statements are a powerful tool to use against SQL Injection attempts. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 25.8 | Cpp:S3649 | Flags SQL queries built using strings. |
| Coverity | 2024.12 | Sql\_injection | Flags unvalidated input in SQL queries. |
| Fortify SCA | 25.3 | SQL Injection | Flags dynamic SQL statements. |
| CodeQL | 2.22.3 | Cpp/sql-injection | Analysis to find SQL Injection vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Avoid Accessing Freed Memory |

| **Noncompliant Code** |
| --- |
| In the following code segment, memory was deleted, but then accessed later on in the program. Once freed, accessing memory in this way can crash a program or cause unexpected behavior that can be exploited. |
| char\* buffer = new char[10];  delete[] buffer;  buffer[0] = ‘a’; |

| **Compliant Code** |
| --- |
| Once cleared, setting buffer to nullptr prevents the ability to access the freed memory. Using a pointer in this way causes a crash, which leads to better debugging the program code to determine if the freed memory is being accessed in a different segment of code when it should not be. |
| char\* buffer = new char[10];  delete[] buffer;  buffer = nullptr; |

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

| **Principles(s):**   * Principle of Fail-Safe Defaults – Access should be denied by default, especially in cases where memory has been freed. Setting the pointer to nullptr accomplishes this. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Cpp.NewDeleteLeaks | Checks memory managed by new/delete. |
| cppcheck | 2.18 | Memleak / useAfterFree | Flags access to memory after it is freed. |
| Coverity | 2024.12 | use\_after\_free | Detects dereference of freed memory. |
| Valgrind | 3.25.1 | memcheck | Flags use after free errors at runtime. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Validate Assumptions with Assertions |

| **Noncompliant Code** |
| --- |
| The following, basic function takes a message as an argument and prints it. If the message passed somehow ends up being null, this can crash the program or cause strange behaviors. It is assumed that the message will never be null, but there is no way to verify this during testing and development. |
| void printMsg(const char\* msg) {  std::cout << msg << std::endl;  } |

| **Compliant Code** |
| --- |
| The revised code block uses an assertion, which should be utilized during testing and development. In the event that a null message is passed, an assertion will catch this, leading to less struggle during troubleshooting. |
| void printMsg(const char\* msg) {  assert(msg != nullptr);  std::cout << msg << std::endl;  } |

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

| **Principles(s):**   * Design by Contract – Ensure the program behaves as intended with use of different types of assertions: preconditions, postconditions, invariants |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.18 | checkassert | Checks for side effects in assert statements. |
| SonarQube | 25.8 | Cpp:S2583 | Flags missing assertions in code. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Always Handle I/O Exceptions |

| **Noncompliant Code** |
| --- |
| The following code segment reads from a file without any exception handling. Any errors that occur from reading the file can crash the program or lead to unintended behavior that can be exploited. |
| void processFile() {  readFile(“config.text”);  }  int main() {  processFile();  } |

| **Compliant Code** |
| --- |
| The revised code segment properly handles any exceptions using a try/catch block for the file read operation. |
| void processFile() {  readFile(“config.text”);  }  int main() {  try {  processFile();  } catch (const std::exception& e) {  std::cout << “Error: “ << e.what() << std::endl;  return EXIT\_FAILURE;  }  } |

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

| **Principles(s):**   * Principle of Fail-Safe Defaults – The program should always handle unexpected problems in safe ways. All potential exceptions should be handled in a consistent manner, every single time. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Hicpp-exception-baseclass | Ensures throw expressions are correct. |
| cppcheck | 2.18 | Exception throw | Flags missing exception handling code. |
| SonarQube | 25.8 | Cpp:S2221 | Flags code lacking exception handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Variables** | [STD-008-CPP] | Initialize Variables on Declaration |

| **Noncompliant Code** |
| --- |
| In the following code segment, a variable is declared but never initialized with a value. In such cases, a line of code may call a variable that has not yet been assigned a value, leading to undefined behavior. |
| int num;  std::cout << “Number: “ << num << std::endl; |

| **Compliant Code** |
| --- |
| In the revised code segment, the declared variable is initialized with a simple value. This prevents undefined behavior and mitigates any errors caused by unintentionally calling a variable with no value. |
| int num = 0;  std::cout << “Number: “ << num << std::endl; |

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

| **Principles(s):**   * Fail-Fast Principle – Ensure that bugs are detected early and quickly. Assigning values to declared variables prevents random, undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | High | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Cppcoreguidelines-init-variables | Checks that variables are initialized with values. |
| cppcheck | 2.18 | Uninitialized variable | Checks if uninitialized variables are used. |
| SonarQube | 25.8 | Cpp:S2755 | Flags uninitialized variables. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Compiler Warnings** | [STD-009-CPP] | Address All Compiler Warnings |

| **Noncompliant Code** |
| --- |
| The following is a code block that will give several warnings from the compiler. Set at a very low level, or suppressed, these warnings will be left undiscovered unless manually corrected by a programmer. |
| int unusedNum = 12; // Compiler Warning: unused variable  int num; // Compiler Warning: uninitialized variable  std::cout << “Number: “ << num << std::endl;  // Compiler Warning: uninitialized variable is used |

| **Compliant Code** |
| --- |
| When the compiler warnings are set to the highest level, the warnings should be addressed and resolved before executing code. |
| int num = 0;  std::cout << “Number: “ << num << std::endl; |

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

| **Principles(s):**   * Clean Code – Develop code that is free of errors. Address all warnings and ensure programs are run error free. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | High | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 25.8 | Cpp:S1126 | Flags code that has ignored compiler warnings. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Concurrency** | [STD-010-CPP] | Avoid Data Races in Concurrent Code |

| **Noncompliant Code** |
| --- |
| In the following example, if the function is called on two separate threads, it can lead to a data race, where the two threads are not properly synchronized as the shared variable is modified. |
| int i = 0;  void counter() {  i++;  } |

| **Compliant Code** |
| --- |
| The mutex library provides a method of preventing data races by preventing the shared variable from being accessed simultaneously between two threads. |
| #include <mutex>  int i = 0;  std::mutex mtx;  void counter() {  std::lock\_guard<std::mutex> lock(mtx);  i++;  } |

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

| **Principles(s):**   * Safety and Predictability – Always protect data in responsible, predictable ways, such as using mutexes to prevent data races. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 22.0 | Thread Safety Analysis | Warns about potential data races in 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.

The first step to automation will be to understand the needs of the program. The needs of the program are to be evaluated and understood well before the design phase, so that security is embedded in the foundation of the program. This security policy, developed in the first step, will be utilized throughout the pre-production and production phases. Static analysis tools will be deployed from the beginning of the design phase, which will frequently been consulted to ensure that security standards are being met, clean code is being developed, and proper testing procedures are being used. These static analysis tools will be run frequently, with results being returned to developers that should be addressed.

### 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 | Medium | Low | High | 3 |
| STD-002-CPP | High | Medium | Low | High | 3 |
| STD-003-CPP | High | Medium | Low | High | 3 |
| STD-004-CPP | High | High | Low | High | 3 |
| STD-005-CPP | High | Medium | Low | High | 3 |
| STD-006-CPP | Medium | Medium | Low | Medium | 2 |
| STD-007-CPP | High | Medium | Low | High | 3 |
| STD-008-CPP | Medium | High | Low | Medium | 2 |
| STD-009-CPP | Medium | High | Low | Medium | 2 |
| STD-010-CPP | High | Medium | Medium | High | 3 |

### 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 at rest | Data stored on hard disks, in databases, and on backups of existing data are encrypted by transforming the data into unreadable code. If physical devices are accessed or stolen, the data remains safe and secure. All sensitive data stored in this manner should be encrypted using the strongest encryption algorithms available, such as AES-256. This policy will apply to any and all data that is saved on any storage medium as listed above. |
| Encryption in flight | Data that is transmitted across networks must be encrypted through the duration of their transfer, using strong methods such as HTTPS or TLS. Data is vulnerable when being transmitted, as attackers are able to intercept the data and read it. If data is properly encrypted, the attackers have a much more difficult time decrypting the data. This policy should be applied whenever any data is being transmitted across a network. |
| Encryption in use | Data that is actively being used is stored in memory and manipulated by the CPU. Proper fail-safes that handle memory leaks should be in effect to prevent arbitrary code from being injected that can corrupt memory, crash systems, or leak the data that is being manipulated. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies that the user attempting to access a system has the valid permissions to access the system. Multi-factor authentication is important, requiring multiple items to be provided from the user, such as a password and an authenticator key. Any login event should employ this policy to prevent access to systems by unauthorized users. |
| Authorization | Authorization determines the permissions that a logged in user is allowed to access. Role-based access control should be used to restrict user access to only what is necessary for them to complete their job functions (principle of least privilege). User access should be regularly reviewed to ensure permissions are correct. |
| Accounting | All user activity should be logged and monitored – logins, data manipulation, file access, and management actions. This ensures that any errors or unauthorized access to a system is quickly logged and reported through the proper channels, allowing correct action to be taken in a reasonable time frame. This helps to track and log security incidents as well. |

**\***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 |  |
| 2.0 | 08/10/2025 | Finalized Template | Brandon Cook |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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