**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 data being inputted into the program needs to be checked to be valid input, and that the input is sanitized to try and prevent SQL injection and other potential threats. |
| 1. Heed Compiler Warnings | Compilers can produce warnings to the programmer of potential issues with the code that may cause the program to not compile further suggesting of a security issue. |
| 1. Architect and Design for Security Policies | Create software architecture and design the software with security policies in mind. From the beginning of design through to the release of product the security policies need to be at the forefront. |
| 1. Keep It Simple | In your program design the simpler the better. More complex programs can increase the likelihood that errors and threats can occur. |
| 1. Default Deny | To manage permissions the best course of action is to deny everyone in the beginning and access management from there. By doing so it allows access to not be granted for fraud. |
| 1. Adhere to the Principle of Least Privilege | Give programs and users only the permissions that they absolutely need and only for as long as needed. This limits what attackers can do. |
| 1. Sanitize Data Sent to Other Systems | Scrub any data sent to other systems (like databases and servers) to avoid attacks like SQL injection. |
| 1. Practice Defense in Depth | Use multiple security measures. If one fails, others can still protect the system. |
| 1. Use Effective Quality Assurance Techniques | Regularly test your code with tools like fuzz testing and penetration testing to find and fix vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Stick to the coding rules designed to improve security for the language and platform that is being used. |

### 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 | Do not declare or define a reserved identifier. |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL51-CPP.+Do+not+declare+or+define+a+reserved+identifier>

| **Noncompliant Code** |
| --- |
| Literal suffix identifiers are required to start with an underscore; literal suffixes without the underscore prefix are reserved for future library implementations. |
| #include <cstddef>    unsigned **int** operator"" x(**const** **char** \*, std::**size\_t**); |

| **Compliant Code** |
| --- |
| The user-defined literal is named operator"" \_x, which is not a reserved identifier. |
| #include <cstddef>    unsigned **int** operator"" \_x(**const** **char** \*, std::**size\_t**); |

| **Principles(s):** Validate Input Data and Adopt a Secure coding standard.  **Justification(s):** Prevents misuse of reserved identifiers, improving code security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | SonarQube’s C++ Analyzer | Detects the use of reserved identifiers, helping to maintain compliance with C++ naming conventions. |
| Clang-Tidy | 15.0 | bugprone-reserved-identifier | Detects the use of identifiers reserved by the C++ Standard. |
| Cppcheck | 2.12 |  | Cppcheck includes checks for naming conventions and can be configured to detect the use of reserved identifiers. |
| PVS-Studio | 7.32 |  | PVS-Studio provides diagnostic rules that can detect the use of reserved identiers, ensuring compliance with C++ standards. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Value-returning functions must return a value from all exit paths |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC52-CPP.+Value-returning+functions+must+return+a+value+from+all+exit+paths>

| **Noncompliant Code** |
| --- |
| The programmer forgot to return the input value for positive input, so not all code paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |

| **Compliant Code** |
| --- |
| All code paths now return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |

| **Principles(s):** Adopt a Secure Coding Standard and Heed Compiler Warnings.  **Justification(s):** Ensuring proper return values and reduces undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | SonarQube’s C++ Analyzer | Detects the use of missing return statements in non-void functions |
| Clang-Tidy | 15.0 | bugprone-missing-return | Provides an extensible framework for diagnosing and fixing typical programming errors, including missing return statements. |
| Coverity Scan | 2024.x |  | Coverity includes checkers that detect missing return statements in value-returning functions. |
| PVS-Studio | 7.32 |  | PVS-Studio provides diagnostics for functions missing return statements, ensuring that all the code paths return a value. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Detect errors when converting a string to a number |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR62-CPP.+Detect+errors+when+converting+a+string+to+a+number>

| **Noncompliant Code** |
| --- |
| Multiple numeric values are converted from the standard input stream. However, if the text received from the standard input stream cannot be converted into a numeric value that can be represented by an int, the resulting value stored into the variables i and j may be unexpected. |
| #include <iostream>    **void** f() {  **int** i, j;    std::cin >> i >> j;    // ...  } |

| **Compliant Code** |
| --- |
| Exceptions are enabled so that any conversion failure results in an exception being thrown. Both the badbit and failbit flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions. |
| #include <iostream>    **void** f() {  **int** i, j;      std::cin.exceptions(std::istream::failbit | std::istream::badbit);  **try** {      std::cin >> i >> j;      // ...    } **catch** (std::istream::failure &E) {      // Handle error    }  } |

| **Principles(s):** Sanitize Data Sent to Other Systems and Validate Input Data.  **Justification(s):** Prevents errors during string conversions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0.0 | clang-analyzer-core.CallAndMessage | Detects potential misuse of string conversion functions. |
| SonarQube | 10.6 | Cpp:S5541 | Detects improper conversion of strings that may lead to undefined behavior. |
| Cppcheck | 2.12 | misra-c2012-21.6 | Detects unsafe string conversion functions. |
| Frama-C | 2022-06-21 | Value Analysis Plugin | Detects potential buffer overflows in string conversions. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CLG | Sanitize data passed to complex subsystems. |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems>

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. An example of an application that inputs an email address to a buffer and then uses this string as an argument to call to system(). |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| A whitelisting approach to data sanitization to define a list of acceptable character and remove any character that is not acceptable. |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

| **Principles(s):** Sanitize Data Sent to Other Systems and Default Deny.  **Justification(s):** Prevents SQL injection by validating and whitelisting inputs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | cpp:S3649 | Detects dynamically constructed SQL queries. |
| OWASP ZAP | 2.12.0 | SQL Injection Scan Rule | Identifies common SQL injection attack vectors. |
| Burp Suite | 2024.1 | Active Scan – SQL Injection | Detects exploitable SQL vulnerabilities. |
| SQLMap | 1.5.12 | Automated SQL Injection Detection | Scans for exploitable SQL injection points. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Properly deallocate dynamically allocated resources. |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources>

| **Noncompliant Code** |
| --- |
| The local variable space is passed as the expression to the placement new operator. The resulting point of that call is then passed to ::operator delete() resulting in undefined behavior. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...    **delete** s1;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...      s1->~S();  } |

| **Principles(s):** Practice Defense in Depth and Use Effective QA Techniques.  **Justification(s):** Prevents memory corruption and leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | Memcheck | Detects improper memory management issues. |
| AddressSanitizer (Asan) | LLVM 15.0 | AddressSanitizer LeakDetector | Identifies memory leaks in C/C++ programs. |
| Dr. Memory | 2.3.0 | Memory Leak Checker | Detects common memory handling issues. |
| Coverity Scan | 2024.x | Memory Resource Management Checker | Detects memory allocation and deallocation inconsistencies. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG | Use a static assertion to test the value of a constant expression |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression>

| **Noncompliant Code** |
| --- |
| Use of the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution. |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

| **Principles(s):** Adopt a Secure Coding Standard and Use Effective QA Techniques.  **Justification(s):** Ensures safe assertions without runtime failures. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | cpp:S6046 | Ensures proper use of static assertions in C++. |
| Clang-Tidy | 15.0.0 | cert-dc103-cpp | Checks for correct usage of static assertions in C++. |
| Cppcheck | 2.12 | static\_assert\_check | Flags improper static assertions. |
| PVS-Studio | 7.32 | V1001 | Detects incorrect use of static\_assert statements. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions>

| **Noncompliant Code** |
| --- |
| Neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| The main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

| **Principles(s):** Practice Defense in Depth and Adhere to Least Privilege.  **Justification(s):** Handles exceptions properly to avoid security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | cpp:S905 | Identifies unhandled exceptions in C++. |
| Clang-Tidy | 15.0.0 | bugprone-exception-escape | Detects functions that may thrown exceptions unexpectedly. |
| Cppcheck | 2.12 | exception-safety-check | Flags improper exception handling practices. |
| CodeSonar | 8.2 | Exception Handling Analysis | Identifies unhandled exceptions and poor exception management. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initializations | STD-008-CPP | Avoid cycles during initialization of static objects |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL56-CPP.+Avoid+cycles+during+initialization+of+static+objects>

| **Noncompliant Code** |
| --- |
| Attempts to implement an efficient factorial function using caching. Because the initialization of the static local array cache involves recursion, the behavior of the function is undefined, even though the recursion is not infinite. |
| #include <stdexcept>    **int** fact(**int** i) noexcept(**false**) {  **if** (i < 0) {      // Negative factorials are undefined.  **throw** std::domain\_error("i must be >= 0");    }    **static** **const** **int** cache[] = {      fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),      fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),      fact(12), fact(13), fact(14), fact(15), fact(16)    };    **if** (i < (**sizeof**(cache) / **sizeof**(**int**))) {  **return** cache[i];    }    **return** i > 0 ? i \* fact(i - 1) : 1;  } |

| **Compliant Code** |
| --- |
| Avoids initializing the static local array cache and instead relies on zero-initialization to determine whether each member of the array has been assigned a value yet and, if not, recursively computes its value. |
| #include <stdexcept>    **int** fact(**int** i) noexcept(**false**) {  **if** (i < 0) {      // Negative factorials are undefined.  **throw** std::domain\_error("i must be >= 0");    }    // Use the lazy-initialized cache.  **static** **int** cache[17];  **if** (i < (**sizeof**(cache) / **sizeof**(**int**))) {  **if** (0 == cache[i]) {        cache[i] = i > 0 ? i \* fact(i - 1) : 1;      }  **return** cache[i];    }    **return** i > 0 ? i \* fact(i - 1) : 1;  } |

| **Principles(s):** Keep it simple and Use Effective QA Techniques.  **Justification(s):** Reduces risk of string-related buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | cpp:S5876 | Detects usage of unsafe string functions like strcpy and springf. |
| Clang-Tidy | 15.0.0 | bugprone-unsafe-functions | Flags unsafe string operations. |
| PVS-Studio | 7.32 | V512 | Detects unsafe string copy functions. |
| Frama-C | 2022-06-21 | Memory Safety Plugin | Detects unsafe string handling. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | STD-009-CLG | Use correct integer precisions |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/c/INT35-C.+Use+correct+integer+precisions>

| **Noncompliant Code** |
| --- |
| To prevent undefined behavior, the function ensures that the argument is less than the number of bits used to store a value of type unsigned int. However, if this code runs on a platform where unsigned int has one or more padding bits, it can still result in values for exp that are too large. |
| unsigned **int** pow2(unsigned **int** **exp**) {  **if** (**exp** >= **sizeof**(unsigned **int**) \* CHAR\_BIT) {      /\* Handle error \*/    }  **return** 1 << **exp**;  } |

| **Compliant Code** |
| --- |
| This solution uses a popcount() function, which counts the number of bits set on any unsigned integer, allowing this code to determine the precision of any integer type, signed or unsigned. |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  **size\_t** popcount(uintmax\_t num) {  **size\_t** precision = 0;  **while** (num != 0) {  **if** (num % 2 == 1) {        precision++;      }      num >>= 1;    }  **return** precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

| **Principles(s):** Practice Defense in Depth and Validate Input Data.  **Justification(s):** Prevents integer overflow and wraparound attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0.0 | clang-analyzer-core.UndefinedBinaryOperatorResult | Detects integer wraparounds |
| PVS-Studio | 7.32 | V1042 | Identifies risky integer operations. |
| Cppcheck | 2.12 | integer-overflow-check | Flags unsafe integer operations. |
| Frama-C | 2022-06-21 | Value Analysis Plugin | Detects integer overflows. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | STD-010-CPP | Do not use std::rand() for generating pseudorandom numbers. |

**Reference:** <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC50-CPP.+Do+not+use+std%3A%3Arand%28%29+for+generating+pseudorandom+numbers>

| **Noncompliant Code** |
| --- |
| The code generates an ID with a numeric part produced by calling the rand() function. The IDs produced are predictable and have limited randomness. Further, depending on the value of RAND\_MAX, the resulting value can have modulo bias. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    id += std::to\_string(std::**rand**() % 10000);    // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the [Mersenne Twister](http://dl.acm.org/citation.cfm?doid=272991.272995) algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the noncompliant code example. |
| #include <random>  #include <string>    **void** f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    std::uniform\_int\_distribution<**int**> distribution(0, 10000);    std::random\_device rd;    std::mt19937 engine(rd());    id += std::to\_string(distribution(engine));    // ...  } |

| **Principles(s):** Architect and Design for Security Policies and Heed Compiler Warnings.  **Justification(s):** Detects and mitigates security vulnerabilities in APIs and inputs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.6 | cpp:S6350 | Detects insecure API usage. |
| OWASP ZAP | 2.12.0 | Input Validation Scanner | Identifies improper input handling. |
| Burb Suite | 2024.1 | Active Scan – API Security | Detects API vulnerabilities. |
| Acunetix | 15.0 | API Security Scanner | Identifies insecure API requests. |

### Defense-in-Depth Illustration

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



### 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 plays a pivotal role in the DevSecOps pipeline, ensuring that security is integrated seamlessly throughout the software development lifecycle (SDLC) without introducing bottlenecks. Traditionally, security was treated as a separate phase in software development, often leading to delays and last-minute vulnerability fixes. However, by embedding security automation into each stage of the DevSecOps pipeline, Green Pace ensures that security checks are performed continuously and efficiently, reducing human error and improving compliance.

In the early phases, automation focuses on static analysis and dependency management. Static Application Security Testing (SAST) tools, such as SonarQube and Coverity, analyze the source code for vulnerabilities before the code is compiled, allowing developers to fix security flaws early. Software Composition Analysis (SCA) tools scan open-source dependencies for known vulnerabilities, preventing supply chain attacks. These automated checks occur within the developer’s IDE or the CI/CD pipeline, ensuring that insecure code is detected before it reaches production.

As the code progresses through the pipeline, Dynamic Application Security Testing (DAST) and Runtime Application Self-Protection (RASP) tools come into play. DAST tools like OWASP ZAP perform security testing by simulating real-world attacks against running applications, identifying vulnerabilities such as SQL injection and cross-site scripting (XSS). RASP solutions monitor applications in real-time and can automatically block suspicious behavior, adding an extra layer of defense. In the final stages of deployment, Integrity Checks and Digital Signing tools ensure that no unauthorized modifications have been made to the software. Post-deployment, Security Information and Event Management (SIEM) systems continuously monitor logs, detect anomalies, and trigger automated incident responses. By leveraging a comprehensive suite of automation tools at every stage, Green Pace effectively enforces security policies, reduces manual intervention, and maintains a secure development environment without sacrificing agility.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Low | P | L3 |
| STD-002-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-003-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-004-CLG | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CLG | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CLG | Low | Unlikely | Medium | P2 | L3 |
| STD-010-CPP | Medium | Unlikely | Low | P6 | L2 |

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | It refers to protecting stored data using cryptographic methods to prevent unauthorized data. It prevents data breaches even if storage devices are stolen or accessed by unauthorized parties. |
| Encryption in flight | Ensures that data moving between systems remains secure and cannot be intercepted. This prevents man-in -the-middle attacks and ensures that sensitive data, such as user credentials, remain protected during transmission. |
| Encryption in use | This refers to encrypting data while it is actively being processed or manipulated in memory. This prevents exposure of sensitive data in memory dumps and mitigates risks of insider threats accessing unencrypted data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This verifies a user’s identity before granting access. It prevents unauthorized access to systems and sensitive data and ensures compliance with security standards like NIST SP 800-63B. |
| Authorization | It determines what resources an authenticated user can access. This prevents privilege escalation attacks and ensures that users and applications only have access to necessary resources. |
| Accounting | This refers to tracking and logging user actions and system activities. It enables forensic analysis of security incidents and support compliance with regulatory frameworks like ISO 27001. |

**\***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 |  |
| 1.1 | 2/11/2025 | Final Project | Caleb Ewer | Caleb Ewer |

## 

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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