**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 | Implement any checks necessary to ensure that data from untrusted sources, particularly from users, is within the design parameters so that it is usable, readable, and secure. This includes checking the data type, size or length, bounds and range, sign, and/or valid characters. |
| 1. Heed Compiler Warnings | Review and correct code highlighted by compiler warnings as they indicate that there are problems in the code that may lead to unexpected behavior during runtime. Unexpected behavior may include vulnerabilities, security flaws, or loss of portability. |
| 1. Architect and Design for Security Policies | Design the code around the established security policies to minimize risk and implement the necessary security strategies. Ensure that the system and components comply with the security policies. |
| 1. Keep It Simple | Design and write code to be as simple as possible without reducing security. Complexity makes code harder to maintain and read, leading to an increased chance of vulnerabilities going unseen. |
| 1. Default Deny | Design the system so that access is denied by default. Permission must be manually granted to each user. |
| 1. Adhere to the Principle of Least Privilege | Only grant each user the privileges that are essential to performing their tasks. |
| 1. Sanitize Data Sent to Other Systems | Modify all data that will be sent to other systems so that malicious or potentially damaging elements are removed. |
| 1. Practice Defense in Depth | Implement multiple, overlapping layers of security. This way, when one layer is breached, the other layers will be able to resist the attack. The layers should synergize to cover the vulnerabilities of each individual layer. |
| 1. Use Effective Quality Assurance Techniques | Adopt techniques such as continuous integration, automation, and test management to properly test and confirm that the system as a whole and individual components meet expectations for function, reliability, security, and performance. |
| 1. Adopt a Secure Coding Standard | Establish the rules for developers to follow when coding that will lead to a secure system. This will lead to a reduction in vulnerabilities and unexpected behavior when the code complies with the rules. |

### 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 delete an array through a pointer of the incorrect type. |
| **Rationalize the Standard** | | Deleting an array through a pointer to the incorrect type results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this example, an array of Derived objects is created and the pointer is stored in a Base \*. Despite Base::~Base() being declared virtual, it still results in undefined behavior. Further, attempting to perform pointer arithmetic on the static type Base \* violates another standard: Do not use pointer arithmetic on polymorphic objects. |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Base \*b = new Derived[10];     // ...     delete [] b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the static type of b is Derived \*, which removes the undefined behavior when indexing into the array as well as when deleting the pointer. |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Derived \*b = new Derived[10];     // ...     delete [] b;  } |

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

| **Principles(s):** Heed Compiler Warnings: A compiler should highlight the non-compliant code due to its incorrect type. Heeding this warning will prevent the vulnerabilities that arise due to undefined behavior being exploited in an attack. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -analyzer-checker=cpluscplus | Checked with clang -ccl or (preferably) scan-build |
| CodeSonar | 9.0p0 | ALLOC.TM | Type Mismatch |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP51-a | Do not treat arrays polymorphically |
| Polyspace Bug Finder | R2024b | CERT C++:EXP51-CPP | Checks for delete operator used to destroy downcast object of different type. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not cast an out-of-range enumeration value |
| **Rationalize the Standard** | | To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. |

| **Noncompliant Code** |
| --- |
| This example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two’s complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);      if (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| The compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    if (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** Heed Compiler Warnings: Compiler and analysis tools should warn of this vulnerability. It is important to maintain security by correcting code which results in unspecified behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Cast-integer-to-enum | Partially checked |
| CodeSonar | 9.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Parasoft C/C++test | 2024.2 | CERT\_CPP\_INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| Polyspace Bug Finder | R2024b | CERT C++:INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator. |
| **Rationalize the Standard** | | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows are a vulnerability which can be exploited to gain access to arbitrary memory locations or to execute arbitrary code. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    void f() {    char buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| To solve this problem, it is best to use std::string instead of a bounded array. |
| #include <iostream>  #include <string>    void f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Validate Input Data & Sanitize Data Sent to Other Systems: Buffer overflows will often result from invalid user input. To protect against such attacks, it is best to validate the data and ensure that a buffer overflow does not occur. It is also important to sanitize the data and ensure that the storage is sufficient for the strings so that we are not sending incomplete data to other systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | MISC.MEM-NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Polyspace Bug Finder | R2024b | CERT C++:STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Sanitize data passed to complex subsystems |
| **Rationalize the Standard** | | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. It is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted. |

| **Noncompliant Code** |
| --- |
| The non-compliant example exploits a vulnerability in the Sun Solaris TELNET daemon (in.telnetd) that allows a remote attacker to log onto the system with elevated privileges. The vulnerability in in.telnetd invokes the login program by calling execl(). This call passes unsanitized data from an untrusted source (the USER environment variable) as an argument to the login program: |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :    getenv("USER")),    0); |

| **Compliant Code** |
| --- |
| The compliant solution inserts the “--" (double dash) argument before the call to getenv (“USER”) in the call to execl(). Because the login program uses the POSIX getopt() function to parse command-line arguments, and because the “--” option causes getopt() to stop interpreting options in the argument list, the USER variable cannot be used by an attacker to inject an additional command-line option. In this context, the resulting string is rendered innocuous. |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :    getenv("USER")), 0); |

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

| **Principles(s):** Sanitize data sent to other systems. By sanitizing the data, this protects against additional arguments being passed to function calls that are passed to other systems. In this case, a potential SQL injection attack can be prevented by preventing SQL commands from being passed into the other system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Not Given | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Avoid using default operator new for over-aligned types |
| **Rationalize the Standard** | | Relying on the default operator new to obtain storage for objects of over-aligned types may result in an object being constructed at a misaligned location, which has undefined behavior and can result in abnormal termination when the object is accessed. |

| **Noncompliant Code** |
| --- |
| In the following example, the new expression is used to invoke the default operator new to obtain storage in which to then construct an object of the user-defined type Vector with alignment that exceeds the fundamental alignment of most implementations (typically 16 bytes). Objects of such over-aligned types are typically required by SIMD (single instruction, multiple data) vectorization instructions, which can trap when passed unsuitably aligned arguments. |
| struct alignas(32) Vector {    char elems[32];  };    Vector \*f() {    Vector \*pv = new Vector;    return pv;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, an overloaded operator new function is defined to obtain appropriately aligned storage by calling the C11 function aligned\_alloc(). |
| #include <cstdlib>  #include <new>    struct alignas(32) Vector {    char elems[32];    static void \*operator new(size\_t nbytes) {      if (void \*p = std::aligned\_alloc(alignof(Vector), nbytes)) {        return p;      }      throw std::bad\_alloc();    }    static void operator delete(void \*p) {      free(p);    }  };    Vector \*f() {    Vector \*pv = new Vector;    return pv;  } |

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

| **Principles(s):** Architect and Design for Security Policies. The mapping between this standard and principle is based on the action of overloading the default operator new to ensure that developers can comply with the security policies when writing code so that memory misalignment does not occur, preventing undefined behavior and abnormal termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MEM57-a | Avoid using the default operator ‘new’ for over-aligned types. |
| Polyspace Buf Finder | R2024b | CERT C++: MEM57-CPP | Checks for situations where operator new is not overloaded for possibly overaligned types (rule fully covered) |
| RuleChecker | 22.10 | Default-new-overaligned-type | Rully checked. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Incorporate diagnostic tests using assertions |
| **Rationalized the Standard** | | Assertions are useful for debugging and will provide details about a failed assert, such as the text of the argument, the name of the source file, the source line number, and the name of the enclosing function. Assertions are also effective when used to protect against incorrect programmer assumptions. |

| **Noncompliant Code** |
| --- |
| The non-compliant example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char \*)malloc(len + 1);    assert(NULL != dup);      memcpy(dup, c\_str, len + 1);    return dup;  } |

| **Compliant Code** |
| --- |
| The compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char\*)malloc(len + 1);    /\* Detect and handle memory allocation error \*/    if (NULL == dup) {        return NULL;    }       memcpy(dup, c\_str, len + 1);    return dup;  } |

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

| **Principles(s):** use Effective Quality Assurance Techniques. Asserts are effective for debugging and are useful for writing diagnostic tests. Diagnostic testing is an important part of effective quality assurance and helps produce better code and can be practiced regularly to ensure that the system is being updated and written with security in mind. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Not Given | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect. |
| Parasoft C/C++test | 2024.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions |
| **Rationalize the Standard** | | A program that encounters an unrecoverable exception may explicitly catch and terminate, but it may not allow the exception to remain uncaught. Catching all exceptions can allow the application to terminate in a controlled fashion. |

| **Noncompliant Code** |
| --- |
| In this example, 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** |
| --- |
| In this solution, 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    }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard: Catching all exceptions should become a standard practice for developers to ensure that the code is more maintainable and readable for others. Uncaught exceptions can lead to unexpected program termination without details to assist in troubleshooting. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Integers** | STD-008-CPP | Ensure that division and remainder operations do not result in divide-by-zero errors. |
| **Rationalize the Standard** | | When the denominator is zero, the operations result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The non-compliant example prevents signed integer overflow but fails to prevent a divide-by-zero error during the division of the signed operands s\_a and s\_b: |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {    signed long result;    if ((s\_a == LONG\_MIN) && (s\_b == -1)) {      /\* Handle error \*/    } else {      result = s\_a / s\_b;    }    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This solution tests the division operation to guarantee there is no possibility of divide-by-zero errors or signed overflow: |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {    signed long result;    if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {      /\* Handle error \*/    } else {      result = s\_a / s\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** Heed Compiler Warnings; Validate Input Data, Sanitize Data Sent to Other Systems. Compiler warnings may highlight code that results in divide-by-zero errors and should be corrected immediately to ensure smooth operation of the program. Validating input data and sanitizing data sent out are also important as untrusted sources can attempt to disrupt the system by manipulating input to result in a divide-by-zero error. Sanitized data can ensure other systems function correctly by removing any possibility of a denominator being zero. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Not Given | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.ARITH.DIVZERO  LANG.ARITH.FDIVZERO | Division by zero  Float Division by Zero |
| Coverity | 2017.07 | DIVIDE\_BY\_ZERI | Fully implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-INT33-a | Avoid division by zero |
| TrustInSoft Analyzer | 1.38 | division\_by\_zero | Exhaustively verified |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input/Output** | STD-009-CPP | Close files when they are no longer needed. |
| **Rationalize the Standard** | | Failing to close files when they are no longer needed can lead to data loss, resource leaks, and conflicts with other programs trying to access the open file. Program crashes may also result in errors saving to the file. |

| **Noncompliant Code** |
| --- |
| In this non-compliant example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| This compliant solution calls std::fstream::close() before std::terminate() gets called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

| **Principles(s):** Architect and Design for Security Policies; Adopt a Secure Coding Standard. It is good practice to ensure all files are closed when they are no longer needed. Incorporating this into the design of the program can ensure that resources are protected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-FIO51-a | Ensure resources are freed. |
| Polyspace Bug Finder | R2024b | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Random Numbers** | STD-010-CPP | Do not use std::rand() for generating pseudorandom numbers. |
| **Rationalize the Standard** | | The function std::rand() makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations have a comparatively short cycle, and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs. |

| **Noncompliant Code** |
| --- |
| The following non-compliant example generates an ID with a numeric part produced by calling the rand() function. The IDs produced are predictable and have limited randomness. |
| #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 solution uses the Mersenne Twister algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the non-compliant 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));    // ...  } |

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

| **Principles(s):** Architect and Design for Security Policies. It is easier to follow other security policies when the code is well-written and written with security in mind from the start. Ensuring that bad functions are ignored and that better functions are used when appropriate, the code base can be inherently made more secure before other layers of defense are implemented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clag | 4.0 (prerelease) | cert-msc50-cpp | Checked by clang-tidy |
| CodeSonar | 9.0p0 | BADFUNC.RANDOM.RAND | Use of rand |
| ÉCLAIR | 1.2 | CC2.MSC30 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |

### 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 is well-suited for pre-production. It is important to identify and fix bugs and errors before pushing a new release to production. The assessment and plan stage should be handled prior to automation to determine what risks and regulatory changes are relevant to the next update. Design, build, and verify and test appear to be the best locations to consider for implementing automation. Test-driven design and best practices apply to the programmer and their approach. Therefore, design is not the ideal stage for automation. Build is also not ideal because this phase is when secure sources are built and used. Thus, the verify and test phase is where automation can most effectively be introduced into the process.

Automation can greatly enhance the Verify and Test phase by assisting with vulnerability scanning and security testing. Many of the automated tools can be incorporated into an IDE or run as a stand-alone program. Each tool has its own specialty and can be used in combination to provide a more holistic scan for vulnerabilities. This is also important for reducing the time spent on identifying vulnerabilities so that more time can be allocated to fixing them. Once fixed, the code can be re-verified to ensure a stronger, more secure release to production.

### 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 | Low | Unlikely | Medium | P2 | L3 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Not Given | P9 | L2 |
| STD-005-CPP | Medium | Unlikely | Low | P6 | L2 |
| STD-006-CPP | Low | Unlikely | Not Given | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Likely | Not Given | P6 | L2 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Medium | Unlikely | Low | P6 | 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 at rest | Encryption at rest means to encrypt and protect data that is stored on a device, such as a server or hard drive. Encryption is a process of converting data into a cipher, which is unreadable without the key necessary to decipher and revert it to the original form. Data at rest is often sensitive information such as a user’s password or credit card number. This policy can be applied in practice by utilizing hashing functions and salting so that the data is unreadable by attackers in the event of a data breach. This should be used as an additional layer of defense so that if other layers fail and an attacker gains access to the system, they cannot make use of the data. |
| Encryption in flight | Encryption in flight means to protect the data that is in transit over a connection to another device. This can include wired and wireless transmission. This type of data is often exchanged between a client and a server. It should be applied in practice by requiring the use of HTTPS or end-to-end encryption. This will ensure that only the intended recipient is able to decrypt the data and that man-in-the-middle attacks are rendered ineffective. This should be used so that users can safely use our service from their browser or via the phone app, even if they are on a public network. |
| Encryption in use | Encryption in use focuses on protecting data that is actively being accessed and modified. The techniques for this type of policy are called partially homomorphic encryption and fully homomorphic encryption. These techniques differ from the others in that the data in-use is encrypted and decrypted in real-time. These are computationally intensive forms of encryption. These techniques prevent processes from outside the protected runtime environment from accessing the protected data. It should be applied in practice for handling highly sensitive data that cannot be isolated from a network. This way, even if the network or device are under attack, the data can remain secure and work on it can continue. It should be used to ensure that valuable data is protected on untrusted machines and networks. |

| 1. **Triple-A Framework\*** | **Explain what it is, how it is used, and why the policy applies.** |
| --- | --- |
| Authentication | Authentication policies are about identifying a user and ensuring that they are who they claim to be. This can include verifying credentials such as user logins. It is used by storing user credentials on a database and then comparing what login details the user provides against the known credentials in our database. The policy applies because an important part of any system utilizing defense in depth is restricting access to specific users. |
| Authorization | Authorization as a policy is about protecting resources so that only those permitted are able to access them. This is used for things like granting permissions for users to complete specific tasks. Each user is granted a specific level of access that is appropriate for their role. In this way, changes to the database are secure and performed only by those with the training and knowledge to do so. Another example is that the addition of new users can only be performed by administrators. This policy applies because defense in depth relies on multiple layers. If a user’s account were to be hijacked, then the authorization policy can limit the damage to what the account has permission to access. This also allows for accounts with access to highly sensitive information to be given additional layers of security rather than applying those additional layers to all employees. |
| Accounting | Accounting as a policy is about logging the actions on the network or service. This includes tracking which resources were accessed, who accessed them, what time, how long, and more. It also includes logging session details for users such as IP address, time of day, duration of session, and device information. It is important to the security of a system to track which files were accessed by users. This allows the security team to track and identify the source of an intrusion and evaluate the scope of damage from an attack. This policy applies because it ensures a swift response to attacks and provides evidence that can be used for updating security policies and defense strategies. |

**\***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 | 06/01/2025 | Completed the description for the 10 coding principles. | Christopher Vargas |  |
| 1.2 | 06/22/2025 | Completed the 10 Coding Standards, Automation paragraph, Encryption Policies, and Triple A policies. | Christopher Vargas |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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