**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 input should be checked and sanitized to prevent potential harm. Ensure all data is correctly checked for format, type, length, and possible exploits from SQL Injections or Buffer Over/Underflow. |
| 1. Heed Compiler Warnings | Pay attention to all warnings from the compilers, as they highlight potential risks and vulnerabilities or unstable code. |
| 1. Architect and Design for Security Policies | Security should be integrated in the beginning of planning and design. There should be clear security policies and a system to enforce them so that security is built into the foundation of the product. |
| 1. Keep It Simple | Code that is simple and direct is less likely to be designed with flaws and vulnerabilities to exploit. Complex code is also harder to understand, test, and maintain. |
| 1. Default Deny | Access should be denied by default and access only granted through explicit permission. |
| 1. Adhere to the Principle of Least Privilege | Users should have minimum access in a system to complete their necessary tasks. Limiting these privileges is damage mitigation that prevents users from accidentally compromising data. |
| 1. Sanitize Data Sent to Other Systems | Data can carry malicious code or vulnerable information, sanitizing output removes potentially harmful code and ensures only the necessary information is sent. Never trust user input. |
| 1. Practice Defense in Depth | Security should have multiple layers so that in case of failure from one layer there is still continuous protection. |
| 1. Use Effective Quality Assurance Techniques | Code should be regularly tested and checked using white box and black box testing procedures, code should be reviewed and sent through automated testing tools. Taking the proper steps ensures security throughout the development lifecycle. |
| 1. Adopt a Secure Coding Standard | Establishing proper protocols and guidelines will ensure consistent security throughout the codebase. This helps to prevent common mistakes that lead to vulnerabilities. |

### 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 define a C-style variadic function |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior. |
| #include <cstdarg>    **int** add(**int** first, **int** second, ...) {  **int** r = first + second;  **va\_list** va;  **va\_start**(va, second);  **while** (**int** v = **va\_arg**(va, **int**)) {      r += v;    }  **va\_end**(va);  **return** r;  } |

| **Compliant Code** |
| --- |
| A variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior. |
| #include <type\_traits>    **template** <**typename** Arg, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Arg s) { **return** f + s; }    **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Ts... rest) {  **return** f + add(rest...);  } |

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

| **Principles(s):** This standard adheres to the principle of type safety by ensuring only integers are passed to the function, preventing undefined behavior and enhancing code reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Function-ellipsis | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL50 |  |
| Clang | 3.9 | cert-dc150-cpp | Checked by clang-tidy |
| CodeSonar | 8.1p0 | LANG.STRUCT.ELLIPSIS | Ellipsis |

#### 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 |

| **Noncompliant Code** |
| --- |
| The function-try-block handler does not return a value, resulting in undefined behavior when an exception is thrown. |
| #include <vector>    std::**size\_t** f(std::vector<**int**> &v, std::**size\_t** s) **try** {    v.resize(s);  **return** s;  } **catch** (...) {  } |

| **Compliant Code** |
| --- |
| The exception handler of the *function-try-block* also returns a value |
| #include <vector>    std::**size\_t** f(std::vector<**int**> &v, std::**size\_t** s) **try** {    v.resize(s);  **return** s;  } **catch** (...) {  **return** 0;  } |

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

| **Principles(s):** Failing to return a value from a code path in a value-returning function results in undefined behavior that might be exploited to cause data integrity violations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | return-implicit | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC52 |  |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as function-try-blocks |
| Helix QAC | 2024.2 | DF2888 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| A std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2024.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-J] | Do not allow tainted variables in privileged blocks |

| **Noncompliant Code** |
| --- |
| This code accepts a tainted path or file name as an argument. An attacker can access a protected file by supplying its path name as an argument to this method. |
| **private** **void** privilegedMethod(**final** String filename)  **throws** FileNotFoundException {  **try** {      FileInputStream fis =          (FileInputStream) AccessController.doPrivileged(  **new** PrivilegedExceptionAction() {  **public** FileInputStream run() **throws** FileNotFoundException {  **return** **new** FileInputStream(filename);          }        }      );      // Do something with the file and then close it    } **catch** (PrivilegedActionException e) {      // Forward to handler    }  } |

| **Compliant Code** |
| --- |
| This solution invokes the cleanAFilenameAndPath() method to sanitize malicious inputs. Successful completion of the sanitization method indicates that the input is acceptable and the doPrivileged() block can be executed. |
| **private** **void** privilegedMethod(**final** String filename)  **throws** FileNotFoundException {  **final** String cleanFilename;  **try** {      cleanFilename = cleanAFilenameAndPath(filename);    } **catch** (/\* exception as per spec of cleanAFileNameAndPath \*/) {      // Log or forward to handler as appropriate based on specification      // of cleanAFilenameAndPath    }  **try** {      FileInputStream fis =          (FileInputStream) AccessController.doPrivileged(  **new** PrivilegedExceptionAction() {  **public** FileInputStream run() **throws** FileNotFoundException {  **return** **new** FileInputStream(cleanFilename);          }        }      );      // Do something with the file and then close it    } **catch** (PrivilegedActionException e) {      // Forward to handler    }  } |

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

| **Principles(s):** Allowing tainted inputs in privileged operations can result in privilege escalation attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | JAVA.IO.PERM.ACCESS  JAVA.IO.PERM | Accessing File in Permissive Mode (Java)  Permissive File Mode (Java) |
| Parasoft Jtest | 2024.1 | **CERT.SEC01.PRIVIL** | Avoid operating on tainted data in privileged blocks |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| In this code, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |

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

| **Principles(s):** Reading previously dynamically allocated memory after it has been deallocated can lead to abnormal program termination and denial-of-service attacks. Writing memory that has been deallocated can lead to the execution of arbitrary code with the permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **dangling\_pointer\_use** |  |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-MEM50** |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.1p0 | **ALLOC.UAF** | Use after free |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| This code uses 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. |
| **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 |

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

| **Principles(s):** Static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities at compile time. The absence of static assertions, however, does not mean that code is incorrect. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ECLAIR | 1.2 | **CC2.DCL03** | Fully implemented |
| LDRA tool suite | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Do not throw an exception across execution boundaries |

| **Noncompliant Code** |
| --- |
| In this code, an exception is thrown from a library function to signal an error. Despite the exception being a scalar type, this code can still result in abnormal program execution if the library and application adhere to different ABIs. |
| // library.h  **void** func() noexcept(**false**); // Implemented by the library    // library.cpp  **void** func() noexcept(**false**) {    // ...  **if** (/\* ... \*/) {  **throw** 42;    }  }    // application.cpp  #include "library.h"    **void** f() {  **try** {      func();    } **catch**(**int** &e) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the error from the library function is indicated by a return value instead of an exception. Using Microsoft Visual Studio (or GCC) to compile both the library and the application would also be a compliant solution because the same exception-handling machinery and ABI would be used on both sides of the execution boundary. |
| // library.h  **int** func() noexcept(**true**); // Implemented by the library    // library.cpp  **int** func() noexcept(**true**) {    // ...  **if** (/\* ... \*/) {  **return** 42;    }    // ...  **return** 0;  }    // application.cpp  #include "library.h"    **void** f() {  **if** (**int** err = func()) {      // Handle error    }  } |

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

| **Principles(s):** The effects of throwing an exception across execution boundaries depends on the implementation details of the exception-handling mechanics. They can range from correct or benign behavior to undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-ERR59-a** | Do not throw an exception across execution boundaries |
| Polyspace Bug Finder | R2024a | CERT C++: ERR59-CPP | Checks for exceptions raised from library interfaces (rule partially covered). |
| Helix QAC | 2024.2 | C++3809, C++3810 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Object Oriented Programming** | [STD-008-CPP] | Do not invoke virtual functions from constructors or destructors |

| **Noncompliant Code** |
| --- |
| In this code, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| **struct** B {    B() { seize(); }  **virtual** ~B() { release(); }    **protected**:  **virtual** **void** seize();  **virtual** **void** release();  };    **struct** D : B {  **virtual** ~D() = **default**;    **protected**:  **void** seize() override {      B::seize();      // Get derived resources...    }    **void** release() override {      // Release derived resources...      B::release();    }  }; |

| **Compliant Code** |
| --- |
| In this solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources. |
| **class** B {  **void** seize\_mine();  **void** release\_mine();    **public**:    B() { seize\_mine(); }  **virtual** ~B() { release\_mine(); }    **protected**:  **virtual** **void** seize() { seize\_mine(); }  **virtual** **void** release() { release\_mine(); }  };    **class** D : **public** B {  **void** seize\_mine();  **void** release\_mine();    **public**:    D() { seize\_mine(); }  **virtual** ~D() { release\_mine(); }    **protected**:  **void** seize() override {      B::seize();      seize\_mine();    }    **void** release() override {      release\_mine();      B::release();    }  }; |

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

| **Principles(s):** Invoking virtual functions during the construction or destruction of an object can lead to unpredictable behavior because the virtual table pointer may not point to the derived class. This undermines the principle of polymorphism and can cause incorrect function calls or incomplete initializations, resulting in potential security vulnerabilities or application crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | **LANG.STRUCT.VCALL\_IN\_CTOR**  **LANG.STRUCT.VCALL\_IN\_DTOR** | Virtual Call in Constructor  Virtual Call in Destructor |
| Helix QAC | 2024.2 | **C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282** |  |
| Klocwork | 2024.2 | **CERT.OOP.CTOR.VIRTUAL\_FUNC** |  |
| LDRA tool suite | 9.7.1 | **467 S, 92 D** | Fully implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Concurrency** | [STD-009-CPP] | Wrap functions that can spuriously wake up in a loop |

| **Noncompliant Code** |
| --- |
| This code monitors a linked list and assigns one thread to consume list elements when the list is nonempty.  This thread pauses execution using wait() and resumes when notified, presumably when the list has elements to be consumed. It is possible for the thread to be notified even if the list is still empty, perhaps because the notifying thread used notify\_all(), which notifies all threads. Notification using notify\_all() is frequently preferred over using notify\_one(). (See CON55-CPP. Preserve thread safety and liveness when using condition variables for more information.)  A condition predicate is typically the negation of the condition expression in the loop. In this noncompliant code example, the condition predicate for removing an element from a linked list is (list->next != nullptr), whereas the condition expression for the while loop condition is (list->next == nullptr).  This code nests the call to wait() inside an if block and consequently fails to check the condition predicate after the notification is received. If the notification was spurious or malicious, the thread would wake up prematurely. |
| #include <condition\_variable>  #include <mutex>    **struct** Node {  **void** \*node;  **struct** Node \*next;  };    **static** Node list;  **static** std::mutex m;  **static** std::condition\_variable condition;    **void** consume\_list\_element(std::condition\_variable &condition) {    std::unique\_lock<std::mutex> lk(m);    **if** (list.next == nullptr) {      condition.wait(lk);    }      // Proceed when condition holds.  } |

| **Compliant Code** |
| --- |
| This solution calls the wait() member function from within a while loop to check the condition both before and after the call to wait(). |
| #include <condition\_variable>  #include <mutex>    **struct** Node {  **void** \*node;  **struct** Node \*next;  };    **static** Node list;  **static** std::mutex m;  **static** std::condition\_variable condition;    **void** consume\_list\_element() {    std::unique\_lock<std::mutex> lk(m);    **while** (list.next == nullptr) {      condition.wait(lk);    }      // Proceed when condition holds.  } |

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

| **Principles(s):** Failure to enclose calls to the wait(), wait\_for(), or wait\_until() member functions inside a while loop can lead to indefinite blocking and denial of service (DoS). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 | **C++5019** |  |
| Klocwork | 2024.2 | **CERT.CONC.WAKE\_IN\_LOOP** |  |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-CON54-a** | Wrap functions that can spuriously wake up in a loop |
| Polyspace Bug Finder | R2024a | CERT C++: CON54-CPP | Checks for situations where functions that can spuriously wake up are not wrapped in loop |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Environment** | [STD-010-C] | Do not call system() |

| **Noncompliant Code** |
| --- |
| In this code example, the system() function is used to execute any\_cmd in the host environment. |
| #include <string.h>  #include <stdlib.h>  #include <stdio.h>    **enum** { BUFFERSIZE = 512 };    **void** func(**const** **char** \*input) {  **char** cmdbuf[BUFFERSIZE];  **int** len\_wanted = snprintf(cmdbuf, BUFFERSIZE,                              "any\_cmd '%s'", input);  **if** (len\_wanted >= BUFFERSIZE) {      /\* Handle error \*/    } **else** **if** (len\_wanted < 0) {      /\* Handle error \*/    } **else** **if** (**system**(cmdbuf) == -1) {      /\* Handle error \*/    }  } |

| **Compliant Code** |
| --- |
| In this solution, the call to system() is replaced with a call to execve(). The exec family of functions does not use a full shell interpreter, so it is not vulnerable to command-injection attacks, such as the one illustrated in the noncompliant code example.  The execlp(), execvp(), and (nonstandard) execvP() functions duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a forward slash character (/). As a result, they should be used without a forward slash character (/) only if the PATH environment variable is set to a safe value, as described in ENV03-C. Sanitize the environment when invoking external programs.  The execl(), execle(), execv(), and execve() functions do not perform path name substitution.  Additionally, precautions should be taken to ensure the external executable cannot be modified by an untrusted user, for example, by ensuring the executable is not writable by the user. |
| #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  #include <errno.h>  #include <stdlib.h>    **void** func(**char** \*input) {    pid\_t pid;  **int** status;    pid\_t ret;  **char** \***const** args[3] = {"any\_exe", input, NULL};  **char** \*\*env;  **extern** **char** \*\*environ;      /\* ... Sanitize arguments ... \*/      pid = fork();  **if** (pid == -1) {      /\* Handle error \*/    } **else** **if** (pid != 0) {  **while** ((ret = waitpid(pid, &status, 0)) == -1) {  **if** (**errno** != EINTR) {          /\* Handle error \*/  **break**;        }      }  **if** ((ret == 0) ||          !(WIFEXITED(status) && !WEXITSTATUS(status))) {        /\* Report unexpected child status \*/      }    } **else** {      /\* ... Initialize env as a sanitized copy of environ ... \*/  **if** (execve("/usr/bin/any\_cmd", args, env) == -1) {        /\* Handle error \*/        \_Exit(127);      }    }  } |

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

| **Principles(s):** If the command string passed to system(), popen(), or other function that invokes a command processor is not fully sanitized, the risk of exploitation is high. In the worst case scenario, an attacker can execute arbitrary system commands on the compromised machine with the privileges of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | **DONT\_CALL** | Implemented |
| Helix QAC | 2024.2 | **C5018**  **C++5031** |  |
| Klocwork | 2024.2 | **SV.CODE\_INJECTION.SHELL\_EXEC** |  |
| LDRA tool suite | 9.7.1 | **588 S** | Fully implemented |

### 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.

Using automation in our security process ensures robust protection at every stage of development, following the DevSecOps model. For planning, we use tools to check for new threats and regulatory changes automatically. During design, automated tools verify that security standards are met. When building, the system scans for vulnerabilities. In testing, we run automatic checks to confirm security and functionality. Deployment involves automated security settings checks. Monitoring systems continuously watch for issues. If problems arise, automated responses block threats and restore safety. After an attack, automated processes help quickly return to normal operations, maintaining security and stability.

This workflow integrates security throughout the pre-production and production phases as shown in the diagram, ensuring continuous and seamless protection from planning to maintenance.

### 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 | Probable | Medium | P12 | L1 |
| STD-002-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-J | High | Likely | Low | P27 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-C | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | High | Probable | Medium | P12 | L1 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-010-C | High | Probable | Medium | P12 | L1 |

### 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 | **What it is**: Encryption at rest protects data stored on physical media such as hard drives or cloud storage by converting it into an unreadable format without the decryption key.  **How it is used**: It ensures that sensitive data remains secure when stored, preventing unauthorized access even if physical security measures are breached.  **Why the policy applies**: This policy applies to all sensitive and personal data stored on Green Pace systems. It is used to comply with data protection regulations and to safeguard against data theft and breaches. |
| Encryption in flight | **What it is**: Encryption in flight secures data while it is being transmitted over networks, making it unreadable to anyone intercepting the communication.  **How it is used**: It uses protocols like TLS (Transport Layer Security) to protect data exchanged between systems, such as between a web server and a browser.  **Why the policy applies**: This policy applies whenever sensitive data is transmitted across networks, including internal and external communications. It ensures data integrity and confidentiality during transfer. |
| Encryption in use | **What it is**: Encryption in use protects data while it is being processed in memory, ensuring that it remains encrypted even when in use by applications.  **How it is used**: It uses techniques like homomorphic encryption to perform computations on encrypted data without decrypting it.  **Why the policy applies**: This policy applies to sensitive data being processed by applications, particularly in environments where there are risks of memory scraping attacks. It ensures continuous data protection during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **What it is**: Authentication verifies the identity of users and systems before granting access to resources.  **How it is used**: Methods include passwords, biometrics, and multi-factor authentication.  **Why the policy applies**: This policy applies to all access points to Green Pace systems to ensure that only authorized users can access sensitive data. It prevents unauthorized access and ensures user accountability. |
| Authorization | **What it is**: Authorization determines what authenticated users are allowed to do, enforcing access controls and permissions.  **How it is used**: Role-based access control (RBAC) and access control lists (ACLs) are common methods.  **Why the policy applies**: This policy applies to all operations on Green Pace systems to ensure that users have the minimum necessary access to perform their tasks. It prevents privilege escalation and data breaches. |
| Accounting | **What it is**: Accounting logs and monitors user activities and system events, providing an audit trail for security analysis.  **How it is used**: Systems record login attempts, file accesses, and changes to user permissions.  **Why the policy applies**: This policy applies to all user and system activities on Green Pace systems. It is crucial for detecting suspicious behavior, forensic analysis, and compliance with regulatory requirements. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.5 | 08/7/2024 | Pre-Approval | Darren Sheftic | Trevor Hodde |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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