**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Coding Standard 8 11](#_heading=h.28h4qwu)

[Coding Standard 9 13](#_heading=h.nmf14n)

[Coding Standard 10 14](#_heading=h.37m2jsg)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## 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.

Seacord, R. (2018, May 2). *Top 10 secure coding practices*. Top 10 Secure Coding Practices - CERT Secure Coding - Confluence. Retrieved November 10, 2022, from [https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices](about:blank)

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | A process of an input matching a specific requirement. This ensures that improper data doesn’t validate. |
| 1. Heed Compiler Warnings | Modifying code based on the highest warning possible for the compiler, this mitigates warnings. By doing so, we reduce vulnerabilities using tools such as static and dynamic. |
| 1. Architect and Design for Security Policies | Implementing and enforcing security policies and making sure they’re in the latest version whenever necessary. This all falls coexists with the creation of software architecture. |
| 1. Keep It Simple | Mitigate complexity when it comes to algorithms and designs as it leads to a higher chance of errors when implementation, configuration, and use takes place. It’s ideal to *keep it simple*. |
| 1. Default Deny | We use permissions for access decisions instead of exclusions. |
| 1. Adhere to the Principle of Least Privilege | Users are given the necessary permissions to execute the job they’re given, nothing more. Otherwise, that will lead to vulnerability and be a target for session hijacking, privilege escalation attacks, etc. from attackers. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data such as databases, COTS, command shells into complex subsystems. |
| 1. Practice Defense in Depth | Protecting a layer with multiple layers of defense just in case if one layer of defense would be exploited, another layer would try to mitigate the threat. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques such as penetration testing helps with identifying what vulnerabilities are present for the hackers to exploit. Fuzz testing helps with this also along with independent security reviews. All these actions ensure the system is more secure against attacks. |
| 1. Adopt a Secure Coding Standard | Development and application of a secure coding standard based on the language and platform that will be 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.

Snavely, W., & Ballman, A. (2018, March 22). *2 rules*. 2 Rules - SEI CERT C++ Coding Standard - Confluence. Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/2+Rules>

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | **STD-001-C++** | **Do not cast to an out-of-range enumeration value** |

| **Noncompliant Code** |
| --- |
| It attempts to check if a given value is within the range of the accepted enumeration values. |
| enum EnumType {  First,  Second,  Third  };    void f(**int** intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This 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. |
| 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):** It is possible for unspecified values to result in a buffer overflow, leading to the execution of arbitrary code by an attacker. However, because enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will result in data integrity violations rather than arbitrary code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | N/A |
| CodeSonar | 7.1p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2022.3 | C++3013 | N/A |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | **STD-002-C++** | **Do not depend on the order of evaluation for side effects** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, *i* is evaluated more than once in an unsequenced manner, so the behavior of the expression is undefined. |
| void f(**int** i, const **int** \*b) {  **int** a = i + b[++i];  // ...  } |

| **Compliant Code** |
| --- |
| This example is independent of the order of evaluation of the operands |
| void f(**int** i, const **int** \*b) {  ++i;  **int** a = i + b[i];  // ...  } |

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

| **Principles(s):** Attempting to modify an object in an unsequenced or indeterminately sequenced evaluation may cause that object to take on an unexpected value, which can lead to unexpected program behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-EXP50 | N/A |
| Clang | 3.9 | -Wunsequenced | Can detect simple violations of this rule where path-sensitive analysis is not required |
| CodeSonar | 7.1p0 | LANG.STRUCT.SE.DEC  LANG.STRUCT.SE.INC | Side Effects in Expression with Decrement  Side Effects in Expression with Increment |
| Compass/ROSE | N/A | N/A | Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | **STD-003-C++** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the unformatted input function std::basic\_istream<T>::read() is used to read an unformatted character array of 32 characters from the given file. However, the read() function does not guarantee that the string will be null terminated, so the subsequent call of the std::string constructor results in undefined behavior if the character array does not contain a null terminator. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  **char** buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }    std::string str(buffer);  // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the std::string object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use std::basic\_istream<T>::readsome() or a formatted input function, depending on need. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  **char** buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }  std::string str(buffer, in.gcount());  // ...  } |

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

| **Principles(s):** Copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute 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** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Helix QAC | 2022.3 | C++2835, C++2836, C++2839, C++5216 | N/A |
| Klocwork | 2022.3 | NNTS.MIGHT  NNTS.TAINTED  NNTS.MUST  SV.UNBOUND\_STRING\_INPUT.CIN | N/A |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | **STD-004-C++** | **Do not access freed memory** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, 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 compliant 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 | 20.10 | dangling\_pointer\_use | N/A |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 | N/A |
| 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 | 7.1p0 | ALLOC.UAF | Use after free |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | **STD-005-C++** | **Honor replacement dynamic storage management requirements** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the global operator new(std::size\_t) function is replaced by a custom implementation. However, the custom implementation fails to honor the behavior required by the function it replaces, as per the C++ Standard, [new.delete.single], paragraph 3. Specifically, if the custom allocator fails to allocate the requested amount of memory, the replacement function returns a null pointer instead of throwing an exception of type std::bad\_alloc. By returning a null pointer instead of throwing, functions relying on the required behavior of operator new(std::size\_t) to throw on memory allocations may instead attempt to dereference a null pointer. See EXP34-C. Do not dereference null pointers for more information. |
| #include <new>    void \*operator new(std::**size\_t** size) {  extern void \*alloc\_mem(std::**size\_t**); // Implemented elsewhere; may return nullptr  return alloc\_mem(size);  }    void operator delete(void \*ptr) noexcept; // Defined elsewhere  void operator delete(void \*ptr, std::**size\_t**) noexcept; // Defined elsewhere |

| **Compliant Code** |
| --- |
| This compliant solution implements the required behavior for the replaced global allocator function by properly throwing a std::bad\_alloc exception when the allocation fails. |
| #include <new>    void \*operator new(std::**size\_t** size) {  extern void \*alloc\_mem(std::**size\_t**); // Implemented elsewhere; may return nullptr  if (void \*ret = alloc\_mem(size)) {  return ret;  }  throw std::bad\_alloc();  }    void operator delete(void \*ptr) noexcept; // Defined elsewhere  void operator delete(void \*ptr, std::**size\_t**) noexcept; // Defined elsewhere |

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

| **Principles(s):** Failing to meet the stated requirements for a replaceable dynamic storage function leads to undefined behavior. The severity of risk depends heavily on the caller of the allocation functions, but in some situations, 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. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2022.3 | C++4736, C++4737, C++4738, C++4739 | N/A |
| Klocwork | 2022.3 | CERT.MEM.OVERRIDE.DELETE  CERT.MEM.OVERRIDE.NEW | N/A |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-MEM55-a | The user defined 'new' operator should throw the 'std::bad\_alloc' exception when the allocation fails |
| Polyspace Bug Finder | R2022b | CERT C++: MEM55-CPP | Checks for replacement allocation/deallocation functions that do not meet requirements of the Standard (rule fully covered) |

#### 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** |
| --- |
| Utilizes 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 |

**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** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 | N/A |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE | N/A | N/A | 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 |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | **STD-007-C++** | **Overload allocation and deallocation functions as a pair in the same scope** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an allocation function is overloaded at global scope. However, the corresponding deallocation function is not declared. Were an object to be allocated with the overloaded allocation function, any attempt to delete the object would result in undefined behavior in violation of MEM51-CPP. Properly deallocate dynamically allocated resources. |
| #include <Windows.h>  #include <new>    void \*operator new(std::**size\_t** size) noexcept(false) {  static **HANDLE** h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }    // No corresponding global delete operator defined. |

| **Compliant Code** |
| --- |
| In this compliant solution, the corresponding deallocation function is also defined at global scope. |
| #include <Windows.h>  #include <new>    class HeapAllocator {  static **HANDLE** h;  static **bool** init;    public:  static void \*alloc(std::**size\_t** size) noexcept(false) {  if (!init) {  h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  init = true;  }    if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }    static void dealloc(void \*ptr) noexcept {  if (h) {  (void)::HeapFree(h, 0, ptr);  }  }  };    **HANDLE** HeapAllocator::h = nullptr;  **bool** HeapAllocator::init = false;    void \*operator new(std::**size\_t** size) noexcept(false) {  return HeapAllocator::alloc(size);  }    void operator delete(void \*ptr) noexcept {  return HeapAllocator::dealloc(ptr);  } |

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

| **Principles(s):** Mismatched usage of new and delete could lead to a denial-of-service attack. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | new-delete-pairwise | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL54 | N/A |
| Clang | 3.9 | misc-new-delete-overloads | Checked with clang-tidy. |
| Helix QAC | 2022.3 | C++2160 | N/A |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | **STD-008-C++** | **A signal handler must be a plain old function** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the signal handler is declared as a static function. However, since all signal handler functions must have C language linkage, and C++ is the default language linkage for functions in C++, calling the signal handler results in undefined behavior. |
| #include <csignal>    static void sig\_handler(**int** sig) {  // Implementation details elided.  }    void install\_signal\_handler() {  if (SIG\_ERR == std::**signal**(SIGTERM, sig\_handler)) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution defines sig\_handler() as having C language linkage. As a consequence of declaring the signal handler with C language linkage, the signal handler will have external linkage rather than internal linkage. |
| #include <csignal>    extern "C" void sig\_handler(**int** sig) {  // Implementation details elided.  }    void install\_signal\_handler() {  if (SIG\_ERR == std::**signal**(SIGTERM, sig\_handler)) {  // Handle error  }  } |

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

| **Principles(s):** Failing to use a plain old function as a signal handler can result in implementation-defined behavior as well as undefined behavior. Given the number of features that exist in C++ that do not also exist in C, the consequences that arise from failure to comply with this rule can range from benign (harmless) behavior to abnormal program termination, or even arbitrary code execution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2022.3 | C++2888 | N/A |
| Klocwork | 2022.3 | CERT.MSC.SIG\_HANDLER.POF | N/A |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-MSC54-a | Properly define signal handlers |
| PRQA QA-C++ | 4.4 | 2888 | N/A |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | **STD-009-C++** | **Preserve thread safety and liveness when using condition variables** |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses five threads that are intended to execute sequentially according to the step level assigned to each thread when it is created (serialized processing). The currentStep variable holds the current step level and is incremented when the respective thread completes. Finally, another thread is signaled so that the next step can be executed. Each thread waits until its step level is ready, and the wait() call is wrapped inside a while loop, in compliance with CON54-CPP. Wrap functions that can spuriously wake up in a loop. |
| #include <condition\_variable>  #include <iostream>  #include <mutex>  #include <thread>    std::mutex mutex;  std::condition\_variable cond;    void run\_step(**size\_t** myStep) {  static **size\_t** currentStep = 0;  std::unique\_lock<std::mutex> lk(mutex);    std::cout << "Thread " << myStep << " has the lock" << std::endl;    while (currentStep != myStep) {  std::cout << "Thread " << myStep << " is sleeping..." << std::endl;  cond.wait(lk);  std::cout << "Thread " << myStep << " woke up" << std::endl;  }    // Do processing...  std::cout << "Thread " << myStep << " is processing..." << std::endl;  currentStep++;    // Signal awaiting task.  cond.notify\_one();    std::cout << "Thread " << myStep << " is exiting..." << std::endl;  }    **int** main() {  constexpr **size\_t** numThreads = 5;  std::thread threads[numThreads];    // Create threads.  for (**size\_t** i = 0; i < numThreads; ++i) {  threads[i] = std::thread(run\_step, i);  }    // Wait for all threads to complete.  for (**size\_t** i = numThreads; i != 0; --i) {  threads[i - 1].join();  }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses notify\_all() to signal all waiting threads instead of a single random thread. Only the run\_step() thread code from the noncompliant code example is modified. |
| #include <condition\_variable>  #include <iostream>  #include <mutex>  #include <thread>    std::mutex mutex;  std::condition\_variable cond;    void run\_step(**size\_t** myStep) {  static **size\_t** currentStep = 0;  std::unique\_lock<std::mutex> lk(mutex);    std::cout << "Thread " << myStep << " has the lock" << std::endl;    while (currentStep != myStep) {  std::cout << "Thread " << myStep << " is sleeping..." << std::endl;  cond.wait(lk);  std::cout << "Thread " << myStep << " woke up" << std::endl;  }    // Do processing ...  std::cout << "Thread " << myStep << " is processing..." << std::endl;  currentStep++;    // Signal ALL waiting tasks.  cond.notify\_all();    std::cout << "Thread " << myStep << " is exiting..." << std::endl;  }    // ... main() unchanged ... |

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

| **Principles(s):** Failing to preserve the thread safety and liveness of a program when using condition variables 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** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | CONCURRENCY.BADFUNC.CNDSIGNAL | Use of Condition Variable Signal |
| Helix QAC | 2022.3 | C++1778, C++1779 | N/A |
| Klocwork | 2022.3 | CERT.CONC.UNSAFE\_COND\_VAR | N/A |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-CON55-a | Do not use the 'notify\_one()' function when multiple threads are waiting on the same condition variable |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | **STD-010-C++** | **Obey the one-definition rule** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two different translation units define a class of the same name with differing definitions. Although the two definitions are functionally equivalent (they both define a class named S with a single, public, nonstatic data member int a), they are not defined using the same sequence of tokens. This code example violates the ODR and results in undefined behavior. |
| // a.cpp  struct S {  **int** a;  };    // b.cpp  class S {  public:  **int** a;  }; |

| **Compliant Code** |
| --- |
| The correct mitigation depends on programmer intent. If the programmer intends for the same class definition to be visible in both translation units because of common usage, the solution is to use a header file to introduce the object into both translation units, as shown in this compliant solution. |
| // S.h  struct S {  **int** a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** Violating the ODR causes undefined behavior, which can result in exploits as well as denial-of-service attacks. As shown in "Support for Whole-Program Analysis and the Verification of the One-Definition Rule in C++" [Quinlan 06], failing to enforce the ODR enables a virtual function pointer attack known as the VPTR exploit. In this exploit, an object's virtual function table is corrupted so that calling a virtual function on the object results in malicious code being executed. See the paper by Quinlan and colleagues for more details. However, note that to introduce the malicious class, the attacker must have access to the system building the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | type-compatibility  definition-duplicate  undefined-extern  undefined-extern-pure-virtual  external-file-spreading  type-file-spreading | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL60 | N/A |
| CodeSonar | 7.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| Helix QAC | 2022.3 | C++1067, C++1509, C++1510 | N/A |

### 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.

A good place to start modifying the existing DevOps process to automate enforcement of the standards in this policy is to take a proactive approach to security. Strong security practices should be instituted throughout the application lifecycle to reduce vulnerabilities, improve security posture and mitigate risks.

*Good DevOps security hygiene practices include:*

* Address security requirements and potential vulnerabilities holistically, since attackers may only need to exploit one vulnerability to carry out their mission.
* Reduce the concentration of privilege in building automation tools and ensure that code repositories do not expose secrets.
* Maintain secrets used by machines and people (passwords, certificates, API keys, tokens, and SSH keys) in a secure, highly available vault—out of source code and off of developer laptops and user-accessible storage systems. Rotate secrets regularly to minimize exposure.

CyberArk. (2022). *DevOps security*. CyberArk. Retrieved from <https://www.cyberark.com/what-is/devops-security/>

### 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Low | P6 | L2 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-010-CPP | High | Unlikely | High | P3 | L3 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Data that’s not being utilized, examples are moving between devices or networks. This type of data is usually stored on hard-drives, laptops, cloud storage, etc. |
| Encryption at flight | Encrypted data is active by movement between devices and networks. Examples of this can include: Moving internally within the organization’s on network or being uploaded to the cloud for backup. |
| Encryption in use | This is where the CPU performs actions in regards to the data such as viewing, coding, or playing a file. If it is being built, modified, or viewed it is considered “in use”. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a method where proof of someone’s identity has to be verified to be granted to the system if they have the proper authorization. There are 5 main authentication factors that we use at the present time. |
| Authorization | Authorization is an act of granting someone access to a confidential file, location, network if they have the proper clearance. Based on certain permissions, this can affect what actions they can perform and what they can do with the data. |
| Accounting | Accounting is having someone taking responsibility for their actions with the clearance they were given along with executing their actions as they were supposed to do. |

**\***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

1. Do not cast to an out-of-range enumeration value (Medium)

Applies as principle #1 because we have to keep the enumeration value in range to prevent error.

1. Do not depend on the order of evaluation for side effects (Medium)

Applies as principle #10 because this is a secure coding standard we can follow.

1. Guarantee that storage for strings has sufficient space for character data and the null terminator (High)

Applies as principle #1 because we have to make sure our strings in code have sufficient storage for their required actions.

1. Do not access freed memory (High)

Applies as principle #10 because memory is volatile so accessing it may disrupt it.

1. Honor replacement dynamic storage management requirements (High)

Applies as principle #10 since dynamic storage management requirements can be beneficial.

1. Use a static assertion to test the value of a constant expression (Low)

Applies as principle #8 because testing values will prove how secure a constant expression is.

1. Overload allocation and deallocation functions as a pair in the same scope (Low)

Applies as principle # 10 because we can adopt this for function to become more secure.

1. A signal handler must be a plain old function (High)

Applies as principle #4 because we just need a signal handler to be a plain old function, nothing more.

1. Preserve thread safety and liveness when using condition variables (Low)

Applies as principle #1 because we input data for a condition variable.

1. Obey the one-definition rule (High)

Applies as principle #4 because having one definition is simple.

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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [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 |