**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 | Validation is the process of confirming that input data is inside the intended range of permitted program input. This means that inputs must comply with the type and numeric range requirements as well , the input invariants of the class or subsystem. |
| 1. Heed Compiler Warnings | Compiler warnings should usually be removed by modifying the code, but if the code is correct, a comment explaining why the warning does not apply is appropriate. Don't just use typeface modifications or other tricks to hide alerts. Rather, understand the rationale behind the warning and come up with a better plan of action, such as matching types and avoiding type casts wherever possible. |
| 1. Architect and Design for Security Policies | Create a software architecture for your program that will enable it to apply and enforce security policies. If, for example, your system requires different privileges at different times, think about dividing it into distinct, communicating subsystems, each with the appropriate set of privileges. |
| 1. Keep It Simple | Try to keep the design as simple and uncomplicated as possible. The likelihood of complex designs being implemented, configured, and used poorly is increased. Furthermore, achieving the appropriate level of assurance requires a great deal more labor as security methods get more complex. |
| 1. Default Deny | Based on authorization, rather than exclusion, make decisions about access. This indicates that access is by default forbidden and that the terms and conditions under which access is permitted are specified by the protection system. |
| 1. Adhere to the Principle of Least Privilege | Each process ought to execute with the minimal rights necessary to do the task at hand. Higher privileges should only be used for the duration required to complete the assigned task. By using this tactic, an attacker's ability to execute arbitrary code with elevated privileges is reduced. |
| 1. Sanitize Data Sent to Other Systems | Make sure that all data transmitted to complex subsystems—such as relational databases, command shells, and commercial off-the-shelf (COTS) components—is cleaned up. Attackers may be able to access unutilized functionality in these components by using SQL, command, or other injection techniques. This is not necessarily a problem with input validation because the complex subsystem being called does not understand the context of the call. Because it knows the context, the calling process is responsible for cleaning the data before starting the subsystem. |
| 1. Practice Defense in Depth | Manage risk by using a range of defensive strategies so that, should one layer of defense prove inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the impact of an effective exploit. For example, employing secure programming techniques in conjunction with secure runtime environments may reduce the likelihood that code vulnerabilities that exist at deployment time will be exploited in the operating environment. |
| 1. Use Effective Quality Assurance Techniques | Sometimes vulnerabilities can be identified and addressed with the use of sound quality assurance practices. Penetration testing, source code audits, and fuzz testing are all essential components of an effective quality assurance program. It's possible that systems that have passed independent security reviews are more secure. For example, external reviewers provide a unique perspective in identifying and rectifying incorrect assumptions. |
| 1. Adopt a Secure Coding Standard | To enhance the security of your software development process, it's crucial to develop and implement a secure coding standard that is specifically tailored to the programming language and platform you are using. This involves identifying a comprehensive set of guidelines and best practices that address common security pitfalls and ensure that code is written in a way that minimizes vulnerabilities. By adopting such a standard, you not only foster a consistent approach to secure coding among your development team but also significantly strengthen the security and reliability of your applications. This proactive step is a fundamental aspect of building a robust security culture within your organization |

### 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** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | There are two types of enumerations in C++: unscoped enumerations, where the underlying type is not fixed, and scoped enumerations, where the underlying type is fixed. Both types of enumerations can represent a range of values, including enumerator values that aren't stated in the enumeration itself. The C++ Standard, [dcl.enum], in paragraph 8, defines the range of acceptable enumeration values for an enumeration type. |

| **Noncompliant Code** |
| --- |
| The goal of this noncompliant code sample is to determine if a given number falls inside the permitted range of enumeration values. It may not be able to represent the supplied integer value, though, because it is casting to the enumeration type first. If a value outside of that range were supplied to f(), the cast to EnumType would result in an ambiguous value, and utilizing that value within the if statement would result in unspecified behavior. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3]. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| In order to ensure that the conversion does not produce an undefined value, this compliant solution first verifies that the value can be represented by the enumeration type. This is accomplished by limiting the converted value to those for whom an enumerator value exists. |
| 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):**   1. **Validate Input Data** - This ensures enumeration values are validated against the defined range before casting, preventing undefined or unexpected behavior. 2. **Architect and Design for Security Policies** - This principle can apply when designing systems to enforce type safety and correctly handle enumeration values. 3. **Keep It Simple** - This rule simplifies handling enumeration values by enforcing type safety and avoiding complex logic that handles out-of-range values. 4. **Adhere to the Principle of Least Privilege** - Limits the allowable operations to those explicitly required, preventing misuse of enumeration casting. 5. **Adopt a Secure Coding Standard**—This rule follows a secure coding practice specific to C++ that helps avoid common pitfalls related to enumeration type conversions, enhancing application security and stability. |
| --- |

**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 |
| Axivion Bauhaus Suite | [Insert text.] | **CertC++-INT50** |  |
| CodeSonar | 8.1p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2024.1 | C++3013 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PVS-Studio | 7.30 | V1016 |  |
| RuleChecker | 22.10 | cast-integer-to-enum | Partially checked |
| Polyspace Bug Finder | R2023b | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **Data Value** | [STD-002-  CPP] | The characteristics design pattern is used by the std::basic\_string type to manage the implementation details of the many string types, producing a set of classes that resemble strings and share an underlying implementation. To be more precise, the std::string, std::wstring, std::u16string, and std::u32string classes are created by pairing the std::basic\_string class with std::char\_traits. To give the std::basic\_string type policy-based implementation details, the std::char\_traits class is specifically specialized. The std::char\_traits::length() method, which is widely used to ascertain the number of characters in a null-terminated string, is one example of an implementation detail. The C++ Standard states that providing a null pointer to this function would result in dereferencing a null pointer, which is undefined behavior [char.traits.require], Table 62 [ISO/IEC 14882-2014]. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code sample, the output of a call to std::getenv() is used to build a std::string object. However, when the environment variable doesn't exist (or some other mistake happens), this method may result in undefined behavior because std::getenv() produces a null pointer on failure. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| Before the std::string object is created in this compliant approach, the outcomes of the call to std::getenv() are examined for null values. |
| #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):**   1. **Validate Input Data** - Check pointers for validity (non-null) before use, preventing crashes or undefined behavior. 2. **Keep It Simple** - This avoids unnecessary complexity and potential bugs by not using null pointers in unsafe ways. 3. **Adopt a Secure Coding Standard** - Follows established guidelines that prevent using null pointers to create **std::string**, reducing security risks and promoting consistency in secure coding practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference Insert text. |
| Helix QAC | 2024.1 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2024.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023b | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Range check element access** |
| --- | --- | --- |
| **String Correctness** | [STD-003-  CPP] | These index operations for std::string operator[]const\_reference(size\_type)  Character held at pos is returned by the const and reference operators[](size\_type). A reference to an object of type charT with value charT() is returned when pos >= size(). The index operators are unchecked (range mistakes do not result in exceptions being fired), and modifying the resulting out-of-range object causes undefinable behavior. |

| **Noncompliant Code** |
| --- |
| The value provided by the get\_index() call in this noncompliant code example might be more than the number of items kept in the string, which would lead to undefined behavior. |
| #include <string>    extern std::size\_t get\_index();    void f() {  std::string s("01234567");  s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| The std::basic\_string::at() method, which functions similarly to the index operator[] but raises a std::out\_of\_range exception if pos >= size(), is used in this complying approach. |
| #include <stdexcept>  #include <string>  **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");  **try** {      s.at(get\_index()) = '1';    } **catch** (std::out\_of\_range &) {      // Handle error    }  } |

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

| **Principles(s):**   1. **Validate Input Data** - Ensuring that index values are within the allowable bounds before accessing array or list elements prevents out-of-bound errors and potential security vulnerabilities. 2. **Keep It Simple** - Keeping the code straightforward by enforcing range checking can prevent errors that might arise from complex and unchecked data manipulations. 3. **Adhere to the Principle of Least Privilege** - This principle ensures that code accesses only the memory locations it is supposed to, adhering to strict access controls. 4. **Practice Defense in Depth** - Implementing multiple layers of checks, including range validations, adds depth to the security measures, guarding against bypasses of single protective layers. 5. **Adopt a Secure Coding Standard** - By adhering to practices that ensure all data accesses are verified, developers maintain high security and reliability standards, reducing the likelihood of exploitable bugs in the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **assert\_failure** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++3162, C++3163, C++3164, C++3165** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR53-a** | Guarantee that container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: STR53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr53cpp.html) | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use valid references, pointers, and iterators to reference elements of a basic\_string** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | This rule represents a particular case of CTR51-CPP since std::basic\_string is a container of characters. To reference a container's elements, use valid references, pointers, and iterators. It is a container that supports iterators in the same way as other Standard Template Library containers. Nevertheless, the invalidation semantics of the std::basic\_string template class are peculiar. The following is stated in the C++ Standard, [string.require], paragraph 5 [ISO/IEC 14882-2014]:  The following uses of that basic\_string object may invalidate references, pointers, and iterators relating to the members of a basic\_string sequence:  as an input parameter for any standard library function that accepts an argument reference for non-const basic\_string.  Calling non-const member functions at, front, back, begin, rbegin, end, and rend; operator[] is the only exception. |

| **Noncompliant Code** |
| --- |
| This example of noncompliant code copies input into a std::string and inserts spaces in place of semicolon (;) characters. Because the iterator loc is invalidated following the initial insert() call, this example is noncompliant. Subsequent insert() calls have unknown behavior. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

| **Compliant Code** |
| --- |
| Every time insert() is called in this conforming approach, the value of the iterator loc is updated, preventing access to the invalidated iterator. The loop ends with an incrementation of the updated iterator. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

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

| **Principles(s):**   1. **Validate Input Data** - Ensuring references, pointers, and iterators are valid before use helps to prevent runtime errors and potential security issues related to accessing invalid memory. 2. **Adhere to the Principle of Least Privilege** - Using only valid and intended pointers or references ensures that the code accesses only the memory it needs, nothing more. 3. **Practice Defense in Depth** - Validating all data handlers, including pointers and iterators, adds an extra layer of security by ensuring that every level of software operation is secure from potential breaches. 4. **Adopt a Secure Coding Standard** - Establishing and following a coding standard that enforces valid data handlers minimizes the risk of security flaws, promoting a more robust and secure application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.UAF** | Use After Free |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **DF4746, DF4747, DF4748, DF4749** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR52-a** | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023b | [CERT C++: STR52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr52cpp.html) | Checks for use of invalid string iterator (rule partially covered). |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-  CPP] | A C++ program can use the memory allocation functions std::malloc(), std::calloc(), and std::realloc(), which are provided by the C programming language. Nevertheless, std::free() is the only method defined by the C programming language for freeing the allocated memory. Refer to MEM31-C. When not in use, free dynamically allocated memory and MEM34-C are provided. Only dynamically allocated free memory for rules that particularly address C allocation and deallocation needs. |

| **Noncompliant Code** |
| --- |
| The local variable space is supplied as the expression to the placement new operator in this example of noncompliant code. After that call, the resultant pointer is handed to::operator delete(), which tries to release memory that was not returned by::operator new(), leading to undefined behavior. |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    delete s1;  } |

| **Compliant Code** |
| --- |
| This compliant solution calls s1's destructor directly rather to removing the call to ::operator destroy(). There are very few situations in which calling a destructor explicitly is appropriate. |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    s1->~S();  } |

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

| **Principles(s):**   1. **Validate Input Data** - Ensuring all dynamic resources are correctly tracked and freed prevents memory leaks and undefined behaviors that can lead to security vulnerabilities. 2. **Adhere to the Principle of Least Privilege** - Deallocating resources that are no longer needed minimizes the attack surface by limiting the time each resource is available. 3. **Practice Defense in Depth** - Proper memory management is a fundamental security practice that adds robustness against exploits that target resource management flaws. 4. **Adopt a Secure Coding Standard** - Following coding standards that include proper resource management can prevent common vulnerabilities related to memory leaks and dangling pointers, enhancing overall software security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **invalid\_dynamic\_memory\_allocation dangling\_pointer\_use** |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MEM51** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Klocwork) | 2024.1 | **CL.FFM.ASSIGN** **CL.FFM.COPY** **CL.FMM** **CL.SHALLOW.ASSIGN** **CL.SHALLOW.COPY** **FMM.MIGHT** **FMM.MUST** **FNH.MIGHT** **FNH.MUST** **FUM.GEN.MIGHT** **FUM.GEN.MUST** **UNINIT.CTOR.MIGHT** **UNINIT.CTOR.MUST** **UNINIT.HEAP.MIGHT** **UNINIT.HEAP.MUST** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MEM51-a** **CERT\_CPP-MEM51-b** **CERT\_CPP-MEM51-c** **CERT\_CPP-MEM51-d** | Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) |  |  | Runtime detection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: MEM51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem51cpp.html) | Checks for:   * Invalid deletion of pointer * Invalid free of pointer * Deallocation of previously deallocated pointer   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.30 | [**V515**](https://pvs-studio.com/en/docs/warnings/v515/), [**V554**](https://pvs-studio.com/en/docs/warnings/v554/), [**V611**](https://pvs-studio.com/en/docs/warnings/v611/), [**V701**](https://pvs-studio.com/en/docs/warnings/v701/), [**V748**](https://pvs-studio.com/en/docs/warnings/v748/), [**V773**](https://pvs-studio.com/en/docs/warnings/v773/), [**V1066**](https://pvs-studio.com/en/docs/warnings/v1066/) |  |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S1232**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-1232) |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-  CPP] | Assertions are an important diagnostic technique for identifying and removing software flaws that could lead to vulnerabilities (see to MSC11-C. Use assertions in diagnostic tests). The  However, the runtime assert() macro has significant drawbacks because it invokes abort() and has a runtime expense. As such, the runtime assert() macro is not helpful for runtime error checking, but rather for detecting false assumptions. Runtime assertions are therefore typically inappropriate for embedded devices or server apps. |

| **Noncompliant Code** |
| --- |
| This noncompliant code asserts a property about a memory-mapped structure that is necessary for the code to function properly using the assert() macro: |
| #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** |
| --- |
| A preprocessor conditional statement can be used for assertions that simply include constant expressions, as demonstrated by this complying 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):**   1. **Validate Input Data**—Static assertions ensure that certain conditions are met at compile time, thus validating assumptions about the program's environment or certain values, leading to safer, more predictable code. 2. **Adopt a Secure Coding Standard** - Using static assertions is part of a secure coding standard that helps detect errors and assumptions during the development phase, preventing potential vulnerabilities from manifesting in the deployed software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL03** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/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](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-  CPP] | In C++, if an exception is thrown and no matching handler is found within the try block or its surroundings in the same thread, the program calls std::terminate(). This function's behavior, like unwinding the stack, varies by implementation. Normally, std::terminate() invokes std::abort(), ending the process abruptly, which can leave resources in uncertain states and might not call destructors for objects. This is a common way denial-of-service attacks happen. It's crucial to catch all exceptions to manage resources and end the process more orderly, even if recovery isn't possible. |

| **Noncompliant Code** |
| --- |
| The exceptions thrown by throwing\_func() are not caught by either f() or main() in this noncompliant code example. Std::terminate() is invoked when there isn't a handler for the thrown exception that matches. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| Since all exceptions are handled at the main entry point in this conforming solution, the stack is guaranteed to unwind up to the main() function, enabling the smooth 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):**   1. **Validate Input Data**—Handling all exceptions is a form of input validation at runtime. This ensures that unexpected data or behavior is managed correctly, maintaining the application's integrity and stability. 2. **Practice Defense in Depth**—By handling exceptions thoroughly, the software implements multiple layers of error handling and control flow management, enhancing the application's robustness and reliability against unexpected operational scenarios. 3. **Use Effective Quality Assurance Techniques** - Proper exception handling is a critical quality assurance practice that helps identify and mitigate issues that may not be evident during initial testing, enhancing the security and functionality of the application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR51** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++4035, C++4036, C++4037** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.1 | **MISRA.CATCH.ALL** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **527 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **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 |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Catch handlers should order their parameter types from most derived to least derived** |
| --- | --- | --- |
| Exception Handling | [STD-008-CCP] | The following is stated in the C++ Standard, [except.handle], paragraph 4 [ISO/IEC 14882-2014]:  A try block's handlers are tried in the order that they occur. This enables the writing of handlers that are never run, such as when a handler for a derived class is placed after a handler for the appropriate base class.  As a result, the most derived exception must be handled first if two handlers catch exceptions that are derived from the same base class (such as std::exception). |

| **Noncompliant Code** |
| --- |
| Because the exceptions of class D are likewise of class B, the first handler in this noncompliant code example catches all exceptions of type B as well. The second handler does not capture any exceptions as a result. |
| // Classes used for exception handling  **class** B {};  **class** D : **public** B {};    **void** f() {  **try** {      // ...    } **catch** (B &b) {      // ...    } **catch** (D &d) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this complying approach, all exceptions of class D are caught by the first handler, while all other exceptions of class B are caught by the second handler. |
| // Classes used for exception handling  **class** B {};  **class** D : **public** B {};    **void** f() {  **try** {      // ...    } **catch** (D &d) {      // ...    } **catch** (B &b) {      // ...    }  } |

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

| **Principles(s):**   1. **Validate Input Data** - Organizing catch handlers from most to least derived ensures that exceptions are matched accurately and handled appropriately. This is a critical component of input validation to maintain program correctness when exceptions occur. 2. **Architect and Design for Security Policies** - Structuring exception handling hierarchically aligns with designing software architectures that enforce security policies by correctly managing exceptions based on their specificity, thus maintaining system integrity under error conditions. 3. **Adhere to the Principle of Least Privilege** - Proper ordering of exception handlers ensures that exceptions are caught and handled at the most specific level necessary, avoiding broader, potentially less secure handling unless needed, thus minimizing the exposed functionality when handling errors. 4. **Practice Defense in Depth** - This approach in exception handling contributes to defense in depth by ensuring that specific errors are caught and handled first, providing layered security measures within error management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **exception-caught-by-earlier-handler** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR54** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -Wexceptions |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ECLAIR) | 1.2 | **CP1.ERR36** | Fully implemented |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++4030, C++4639** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.1 | **MISRA.CATCH.NOALL** **MISRA.CATCH.WRONGORD** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **541 S, 556 S** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR54-a** | Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: ERR54-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr54cpp.html) | Checks for:   * Exception handlers not ordered from most-derived to base class * Incorrect order of ellipsis handler   Rule fully covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.30 | [**V759**](https://pvs-studio.com/en/docs/warnings/v759/) |  |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **exception-caught-by-earlier-handler** | Fully checked |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S1045**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-1045) |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input Output (FIO) | [STD-009-CCP] | A call to std::basic\_filebuf::close() must follow a call to std::basic\_filebuf::open() before either the caller's return value is stored in a reference that expires or the program terminates normally, whichever comes first.  std::basic\_ifstream, std::basic\_ofstream, and std::basic\_fstream all keep an internal reference to an instance of std::basic\_filebuf, on which calls to open() and close() are made when required. To achieve compliance with this rule, an object of one of these types must be managed appropriately, i.e., not leaked. To ensure compliance with MEM51-CPP, it is often desirable to use the stream object by value semantics rather than via dynamic memory allocation. Deallocate dynamically allocated resources appropriately. That is still insufficient, though, for circumstances where destructors are not automatically called. |

| **Noncompliant Code** |
| --- |
| This example of noncompliant code creates an object file called std::fstream. Std::basic\_filebuf<T>::open() is called by the constructor for std::fstream, and std::abort(), the default std::terminate\_handler called by std::terminate(), does not call destructors. As a result, the object is not closing the underlying std::basic\_filebuf<T> object correctly. |
| #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** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is 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):**   1. **Validate Input Data**: Closing files ensures all file operations are finalized, preventing data corruption. 2. **Keep It Simple**: Simplifies resource management by reducing the complexity of open resource tracking. 3. **Adhere to the Principle of Least Privilege**: Limited duration resources are open, reducing exposure. 4. **Practice Defense in Depth**: Protects against potential data leaks or unauthorized access by ensuring resources are not left unnecessarily available. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.LEAK** | Leak |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **DF4786, DF4787, DF4788** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.1 | **RH.LEAK** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) |  |  | Runtime detection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: FIO51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio51cpp.html) | Checks for resource leak (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not slice derived objects** |
| --- | --- | --- |
| Object Oriented Programming | [STD-010-CCP] | Additional member variables that expand the base class are usually included in an object that derives from a base class. Those extra member variables are not duplicated when by-value assigning or copying an object of the derived type to an object of the base type because there is not enough room in the base class to hold them. Because the extra members are "sliced off" of the final object, this procedure is frequently referred to as "slicing the object."  A base class object should never be initialized with a derived class object unless references, pointers, or pointer-like abstractions (like std::shared\_ptr or unique\_ptr) are used. |

| **Noncompliant Code** |
| --- |
| An object of the derived Manager type is supplied by value to a function that accepts a base Employee type in this example of noncompliant code. As a result, when the print() function is called, the Manager objects are sliced, leading to data loss and unpredictable behavior. |
| #include <iostream>  #include <string>    class Employee {  std::string name;    protected:  virtual void print(std::ostream &os) const {  os << "Employee: " << get\_name() << std::endl;  }    public:  Employee(const std::string &name) : name(name) {}  const std::string &get\_name() const { return name; }  friend std::ostream &operator<<(std::ostream &os, const Employee &e) {  e.print(os);  return os;  }  };    class Manager : public Employee {  Employee assistant;    protected:  void print(std::ostream &os) const override {  os << "Manager: " << get\_name() << std::endl;  os << "Assistant: " << std::endl << "\t" << get\_assistant() << std::endl;  }    public:  Manager(const std::string &name, const Employee &assistant) : Employee(name), assistant(assistant) {}  const Employee &get\_assistant() const { return assistant; }  };    void f(Employee e) {  std::cout << e;  }    int main() {  Employee coder("Joe Smith");  Employee typist("Bill Jones");  Manager designer("Jane Doe", typist);    f(coder);  f(typist);  f(designer);  } |

| **Compliant Code** |
| --- |
| Using the same class definitions as the noncompliant code example, this compliant solution modifies the definition of f() to require raw pointers to the object, removing the slicing problem. |
| // Remainder of code unchanged...    void f(const Employee \*e) {  if (e) {  std::cout << \*e;  }  }    int main() {  Employee coder("Joe Smith");  Employee typist("Bill Jones");  Manager designer("Jane Doe", typist);    f(&coder);  f(&typist);  f(&designer);  } |

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

| **Principles(s):**   1. **Keep It Simple**: Prevents errors that could complicate the codebase by ensuring type integrity and avoiding bugs related to object slicing. 2. **Default Deny**: Enforces strict type adherence, preventing any unintended behavior or data access issues arising from implicit type conversions. 3. **Adopt a Secure Coding Standard**: This standard encourages the enforcement of strong type safety rules, reducing the chance of runtime errors and security vulnerabilities associated with object slicing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.CAST.OBJSLICE** | Object Slicing |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++3072** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-OOP51-a** | Avoid slicing function arguments / return value |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: OOP51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcoop51cpp.html) | Checks for object slicing (rule partially covered) |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.30 | [**V1054**](https://pvs-studio.com/en/docs/warnings/v1054/) |  |

### 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 to enforce and ensure compliance with 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 the standards in this policy. Use the DevSecOps diagram to provide an explanation.

Automation across the DevSecOps lifecycle is critical to optimizing Green Pace's well-established DevOps methodology for improved compliance and enforcement of security standards. Automation tools such as Assess and Plan should be used in the early stages to carry out pre-development security evaluations and create baseline security criteria that serve as a roadmap for all subsequent development stages. Static and dynamic application security testing (SAST/DAST) and other security tests should be automatically incorporated into continuous integration and delivery pipelines as the project moves into the DEV, SEC, and OPS stages. This minimizes vulnerabilities early on by ensuring that security is ingrained in the product from the first line of code.

Automation should be used later in the lifecycle, in the Maintain and Stabilize and Monitor and Detect phases, to manage security patches and regularly check the system's health. After deployment, this is essential for preserving the security and integrity of programs. Real-time threat alerts from automated monitoring systems will enable prompt response and minimize possible harm to the team. Automation should assist quick response plans and verify security measures before deployment in crucial phases like Respond, Verify, and Test. This all-encompassing use of automation improves operational effectiveness and reduces human error, but it also fortifies security throughout the development process, guaranteeing a solid defense against possible intrusions.

### 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 | High | Likely | Medium | P18 | L1 |
| STD-003-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-004-CPP | High | Probable | High | P6 | L2 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Medium | Likely | Low | P18 | L1 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Low | Probable | Medium | P4 | 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 at rest | * **What it is**: Encryption at rest involves encrypting data that is stored on physical media, ensuring that data is unreadable without decryption keys. * **Application and Importance**: This policy should be applied to all sensitive data stored on servers, backups, and other storage devices. It protects against data theft from stolen or inappropriately accessed storage devices. Using strong encryption methods like AES ensures that data is protected even if physical security is breached. |
| Encryption in flight | * **What it is**: Encryption in flight refers to encrypting data while it is being transmitted across networks. * **Application and Importance**: This policy should be implemented for all data exchanges over untrusted networks, including the internet. Utilizing protocols such as TLS/SSL for web traffic, SSH for remote connections, and VPNs for secure network access ensures data integrity and confidentiality. This prevents data interception and unauthorized disclosure during transmission |
| Encryption in use | * **What it is**: Encryption in use involves encrypting data actively being processed in memory. * **Application and Importance**: Apply this policy when handling highly sensitive data that must be protected against runtime attacks such as cold boot attacks and side-channel attacks. Techniques include using Trusted Execution Environments (TEEs) and homomorphic encryption. It allows sensitive data to be processed securely, even in compromised environments. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | * **What it is**: Authentication verifies the identities of users, systems, or entities before allowing access to system resources. * **Application and Importance**: It is confirmed that the user is an authorized system user during the authentication process. This could contain the user's password and login information for specific system components. Several more modern methods employ two-step or multi-tier authentication. |
| Authorization | * **What it is**: Authorization involves granting or denying rights and permissions to authenticated entities based on security policies. * **Application and Importance**: A user's access level within a system is determined by their authorization, which also sets restrictions on actions including reading, creating, modifying, and deleting database files. In order to guarantee that every user has access that is appropriate for their roles and responsibilities, it also controls the authority of a user to manage files and other users. This organized access control keeps sensitive data safe inside the system and aids in preventing illegal activities. |
| Accounting | * **What it is**: Accounting tracks and logs user activities within the system, particularly those that affect important data or system configurations. * **Application and Importance**: Accounting, as it relates to system security, is the tracking and observation of user behavior according to access levels. This entails keeping tabs on the databases individuals access, the precise actions they take when logged in, and who is launching these actions. Ensuring accountability, auditing user activity, and strengthening security measures against unwanted data tampering all depend on this kind of thorough tracking. |

\*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 | 3/22/2024 | 1st Revised | Blake Kemp |  |
| 1.2 | 4/12/2024 | Final Revision | Blake Kemp |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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