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



# Green Pace Secure Development Policy

# Mark Eilers

## 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 | Validate Input Data is used to ensure that the data entered into the workflow is correctly done. By doing this, this prevent any malformed data from entering the database. Any potential data that is untrustworthy should be processed by validating the users input. |
| 1. Heed Compiler Warnings | The compliers are created to warn developers of potential errors when writing and designing code in programming. Heeding these warning is to actively acknowledge, fix, and improve the code. The purpose of the heed complier warnings is to provide assistance to the programmer in identifying easy to miss mistakes. |
| 1. Architect and Design for Security Policies | Architect and design for security policies involved designated tools which will describe necessary security countermeasures that can be taken. A security policy statement outlines how entities interact with one another. This can also include how the entities operate. Additionally, outlined in the policy are the levels of protection and the actions that need to be taken when there is a lack of requirements met. |
| 1. Keep It Simple | Keep it simple is the idea that when creating systems and designs, the program should be kept as simples as possible. This will prevent complexity in code that might make the overall design convoluted. |
| 1. Default Deny | Default deny is a principal that states unless an action or ability is allowed, then it should be denied. By doing this, it prevents possible actions from being performed that are unintended and possibly malicious. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the principal of least privilege means that the user is given the least number of permissions that are necessary. In other words, a user should not be given more permissions that was in necessary for their intended purposes. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems refers to the practice of “sanitizing” or “cleaning” data. This means that unwanted characters is removed before the data us processed. If and when a trust boundary is crossed by data, it should be “cleaned” before it is allowed to cross that “boundary”. |
| 1. Practice Defense in Depth | Practice defense in depth is the practice where developers have created multiple, unrelated, layers of security for a single attack. The principal is that if one layer of security fails at stopping the malicious attempt, there are other defending layers as back up to help defend. |
| 1. Use Effective Quality Assurance Techniques | Use effective quality assurance techniques is the practice where developers use the effective techniques to ensure proper function of code in both purpose and security. There are many different techniques that can be used for QA techniques. Some of these include: testing early and often, communicate rules, and outline requirements. |
| 1. Adopt a Secure Coding Standard | Secure coding standards are a set of rules and guidelines that help the developer prevent vulnerabilities in their program. Some examples of securing coding standards include CERT, DISA, and CVE. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not cast to an out-of-rand enumeration value. |

| **Noncompliant Code** |
| --- |
| This example endeavors to verify if a provided value falls within the acceptable range of enumeration values, yet it contains a crucial error. The mistake lies in assigning the enumeration type before checking if it is within the specified range. |
| 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 this compliant code, the verification ensures that the value is representable by the enumeration type before proceeding with the conversion. This precautionary step ensures that the conversion does not yield an unspecified value. This is achieved by limiting the converted value to one that corresponds to a specific enumerator value. |
| Enum EnumType {  First,  Second,  Third  };  Void f(int intVar) {  if (enumVar < First || enumVar > Third) {  // Handle Error  }  EnumType enumVar = static\_cast <EnumTypes>(intVar);  } |

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

| **Principles(s):** Keep it simple – When designing code, it is important to prevent complications and/or errors in calculations. It is imperative to use assigned integers and buffers in a straightforward and concise manner. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | Cast-integer-to-enum | Astrée version 22.10 specializes in detecting and addressing issues related to casting integers to enums, aiming to ensure type safety and prevent potential runtime errors in the software. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.5p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | CodeSonar version 7.5p0 focuses on detecting and resolving issues related to coercion and value casting in the language, aiming to ensure type safety and prevent potential runtime errors in the software. |
| [Parasoft C/C+ + test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-INT50-a | Parasoft C/C++ Test version 2023.1 targets the CERT\_CPP-INT50-a guideline, likely focusing on detecting and resolving issues related to integer conversions in C++ code to ensure compliance with established standards and improve code reliability. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | Cast-integer-to-enum | RuleChecker version 22.10 specializes in identifying and rectifying instances of casting integers to enums, aiming to enhance type safety and prevent potential runtime errors in software implementations. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Utilize valid references, pointers, and iterators to access elements within a container. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example that does not comply with standards, the variable "pos" becomes invalidated after the initial call to insert(), leading to undefined behavior in subsequent loop iterations. |
| #include <deque>  Void f(const double \*items, std::size\_t count) {  std::deque<double> d;  auto pos = d.begin();  for (std::size\_t I = 0; I < count; ++I, ++pos) {  d.insert(pos, items[i] + 41.0);  }  } |

| **Compliant Code** |
| --- |
| In this compliant code that adheres to standards, a valid iterator is assigned to "pos" during each insertion, thus avoiding any undefined behavior. |
| #include <deque>  Void f(const double \*items, std::size\_t count) {  std::deque<double> d;  auto pos = d.begin();  for (std::size\_t i = 0; I < count; ++i, ++pos) {  pos = d.insert(pos, items[i] + 41.0);  }  } |

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

| **Principles(s):** Architect and Design for Security Policies - By carefully structuring and strategizing the utilization of memory in our program, we can preempt typical errors and enhance its security. By strictly adhering to proper access protocols, we sidestep potential vulnerabilities and uphold security standards. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | Overflow\_upon\_derefernce | Astrée version 22.10 is focused on detecting and addressing issues related to potential buffer overflow upon dereferencing, aiming to prevent memory corruption and enhance software security. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | ALLOC.UAF | CodeSonar version 7.4p0 targets detecting and resolving issues related to use-after-free errors in memory allocation, aiming to enhance software reliability and prevent potential security vulnerabilities. |
| [Parasoft C/C+ + test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP\_CTR51-a | Parasoft C/C++ Test version 2023.1 focuses on detecting and resolving issues related to the CERT\_CPP\_CTR51-a guideline, likely concerning proper initialization of objects in C++ code to ensure compliance with coding standards and improve software reliability. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C++ : CTR51-CPP | Polyspace Bug Finder version R2023a targets the CERT C++ CTR51-CPP guideline, emphasizing detection and resolution of issues related to proper initialization of objects in C++ code to enhance software reliability and compliance with coding standards. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensure that the storage allocated for strings has adequate space to accommodate both the character data and the null terminator, thereby preventing buffer overflow. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, exceeding 12 characters in a single character array will trigger a buffer overflow. Although employing std::ios\_base::width() can assist in restricting the number of characters allowed in bufOne, it's worth noting that a truncated string may still be stored in bufOne, potentially leading to overflow in the second character array. |
| #include <iostream>    void f() {  char bufOne[12];  char bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo;  } |

| **Compliant Code** |
| --- |
| For the compliant code, to mitigate the risk of potential buffer overflow, a straightforward approach is to utilize std::string instead of a bounded char array. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Validate Input Data - aiming to ensure that strings passed into the program are properly null terminated. Failure to null terminate strings can result in functions that rely on this termination failing or causing undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | STRING\_SIZE | Coverity version 2017.07 focuses on detecting and addressing issues related to incorrect string size handling in the code, aiming to improve software reliability and security. |
| Parasoft C/C++ test | 2020.2 | CERT\_C-STR31-e | Parasoft C/C++ Test version 2020.2 targets the CERT\_C-STR31-e guideline, likely focusing on detecting and resolving issues related to improper string manipulation in C code to ensure compliance with coding standards and improve software reliability. |
| PVS-Studio | 7.07 | V518, V645, V727, V755 | PVS-Studio version 7.07 is geared towards detecting and resolving issues related to specific error codes (V518, V645, V727, V755) to enhance code quality and identify potential vulnerabilities. |
| TrustInSoft Analyzer | 1.38 | Meme\_access | TrustInSoft Analyzer version 1.38 specializes in identifying and resolving issues related to memory access errors, aiming to enhance code safety and reliability. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Avoid storing a pointer value that is already owned within an unrelated smart pointer. |

| **Noncompliant Code** |
| --- |
| The following noncompliant example features two unrelated smart pointers constructed from the same underlying pointer value. Upon the destruction of the local automatic variable p2, it deletes the managed pointer value. Subsequently, upon the destruction of the local automatic variable p1, it also deletes the same pointer value, leading to a double-free vulnerability. |
| #include <memory>  void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| The compliant code involves two std::shared\_ptr objects that are interconnected through copy construction. Upon the destruction of the local automatic variable p2, the use count for the shared pointer value is decremented but remains nonzero. Subsequently, upon the destruction of the local automatic variable p1, the use count for the shared pointer value is decremented to zero, leading to the destruction of the managed pointer. This compliant solution also utilizes std::make\_shared() instead of directly allocating a raw pointer and storing its value in a local variable. |
| #include <memory>  void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):** Validate Input Data - Ensuring the validation of input data serves to block malware from infiltrating the system and verifies whether the data is suitable for processing securely. |
| --- |

**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 | Dangling\_pointer\_use | "Dangling\_pointer\_use" is one such issue it identifies, where a pointer is used after the memory it points to has been deallocated, leading to potential security risks. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | CertC++-MEM56 | Axivion Bauhaus Suite, version 7.2.0, specifically addresses the CertC++-MEM56 guideline, which likely pertains to memory management practices in C++ code, ensuring compliance with established standards and enhancing code reliability. |
| [Parasoft C/C+ + test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-MEM56-a | Parasoft C/C++ Test version 2023.1 targets the CERT\_CPP-MEM56-a guideline, likely focusing on memory management practices in C++ code to ensure adherence to established standards and improve code robustness. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | CERTC++:Mem56-CPP | Polyspace Bug Finder version R2023a addresses the CERTC++:Mem56-CPP guideline, indicating a focus on detecting and resolving memory management issues in C++ code to enhance software reliability and security. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | When employing malloc/alloc, ensure the detection and proper handling of memory allocation errors. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, it fails to verify the success of memory allocation before proceeding with further operations. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = (int\*) malloc(5\*sizeof(int));  delete [] copy;  } |

| **Compliant Code** |
| --- |
| In the compliant code, we verify the presence of a valid memory address within the pointer, we adhere to the standard. |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = (int\*) malloc(5\*sizeof(int));  if (!copy) {  // Handle error  return;  }  delete [] copy;  } |

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

| **Principles(s):** Validate Input Data - Although accessing freed memory typically leads to an application crash, there are instances where it may not happen immediately, potentially allowing access to unintended memory or overwriting values of other stored variables. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P4 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | leakReturnValNotUsed | Cppcheck version 1.66 focuses on detecting and addressing issues related to memory leaks where returned values are not utilized, aiming to enhance code reliability and resource management. |
| Parasoft C/C++test | 2020.2 | CERT\_C-MEM31-a | Parasoft C/C++test version 2020.2 targets the CERT\_C-MEM31-a guideline, likely focusing on detecting and resolving issues related to improper memory deallocation in C code to ensure compliance with coding standards and improve software reliability. |
| SonarQube C/C++ Plugin | 3.11 | S3584 | SonarQube C/C++ Plugin version 3.11 focuses on detecting and addressing issues related to potential memory leaks, aiming to enhance code reliability and resource management. |
| TrustInSoft Analyzer | 1.38 | malloc | TrustInSoft Analyzer version 1.38 specializes in analyzing the behavior of malloc, aiming to ensure memory safety and prevent vulnerabilities in software code. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Employ a static assertion to evaluate the value of a constant expression. |

| **Noncompliant Code** |
| --- |
| The noncompliant code utilizes the assert() macro to assert a property regarding a memory-mapped structure crucial for the correct behavior of the code. |
| #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** |
| --- |
| In this compliant code, the assertion involves solely a constant expression. Therefore, a preprocessor conditional statement can be utilized, as demonstrated in the solution. |
| #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));  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques & Keep it Simple - Employing efficient quality assurance methods is crucial; catching issues early prevents their escalation. Utilizing static assertions offers developers a means of quality assurance, potentially eradicating defects. Simplify whenever possible; avoid unnecessary complexity in the program. Implement or utilize assertions in their simplest forms to uncover hidden code problems. |
| --- |

**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 | Axivion Bauhaus Suite version 7.2.0 targets the CertC-DCL03 guideline, likely focusing on detecting and addressing issues related to declaration and initialization in C code to ensure code quality and compliance with coding standards. |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | Misc-static-assert | Clang version 3.9 addresses the Misc-static-assert guideline, suggesting a focus on static assertion checks to improve code quality and identify potential issues during compilation. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | Customization | CodeSonar version 7.4p0 emphasizes customization, indicating a focus on tailoring the analysis process to specific project requirements or coding standards for enhanced accuracy and efficiency. |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC3.DCL03 | ECLAIR version 1.2 targets the CC3.DCL03 guideline, suggesting a focus on detecting and resolving declaration-related issues in C code to ensure compliance with coding standards and improve code quality. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STR-007-CPP] | Exceptions provide an effective mechanism to prompt calling code to acknowledge an error condition and determine how to manage it. Unhandled exceptions halt program execution, thereby ensuring robust handling of error conditions and preventing security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This noncompliant example, while incorporating exceptions, lacks optimal utility. Throwing a generic exception fails to provide the programmer with information about the expected exception to be caught and how to appropriately handle the situation. |
| void raise\_exception() noexcept(false);    void target() {  raise\_exception();  }    int main() {  target();  } |

| **Compliant Code** |
| --- |
| This complaint code effectively specifies the expected exception from this method, enabling a programmer to appropriately handle this scenario if it arises. |
| void raise\_exception() noexcept(false);    void target() {  raise\_exception();  }    int main() {  try {  target();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** Develop a Coding Standard - Failure to handle all exceptions may not affect an application under normal circumstances, but any unhandled exceptions arising from erroneous data can lead to application crashes and data corruption. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | P10 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | main-function-catch-all  early-catch-all | Astrée version 20.10 specializes in detecting and addressing issues related to early catch-all or main function catch-all constructs, aiming to ensure code reliability and robustness. |
| LDRA tool suite | 9.7.1 | 527 S | LDRA Tool Suite version 9.7.1 focuses on detecting and resolving issues related to rule 527 S, aiming to improve code quality and compliance with coding standards. |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Parasoft C/C++test version 2020.2 targets the CERT\_CPP-ERR51-a and CERT\_CPP-ERR51-b guidelines, likely focusing on detecting and resolving issues related to error handling in C++ code to ensure compliance with established standards and improve software reliability. |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | RuleChecker version 20.10 specializes in detecting and resolving issues related to early catch-all or main function catch-all constructs, aiming to ensure code reliability and robustness. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | When handling exceptions, be careful to no leak resources. |

| **Noncompliant Code** |
| --- |
| This non-compliant example is problematic due to its error handling, which fails to properly clean up the allocated myVector when an operation is thrown. |
| #include <vector>  void f() {  std::vector<int> \*myVector = new std::vector<int>(5);  try  {  myVector->at(1)  } catch (…) {  // Process error with logic  throw;  }  delete myVector;  } |

| **Compliant Code** |
| --- |
| This code adheres to the standard, as the try/catch block ensures the destruction of the allocated vector, regardless of whether an error is thrown from at(1). This is achieved by employing a finally block associated with the try catch. |
| #include <vector>  void f() {  std::vector<int> \*myVector = new std::vector<int>(5);  try  {  myVector->at(1)  } catch (…) {  // Process error with logic  throw;  }  finally  {  // Handle the deletion of the allocated memory  delete myVector;  }  } |

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

| **Principles(s):** Heed Complier Warnings/ Use Effective Quality Assurance Techniques - Paying attention to compiler warnings in this case will help prevent such issues and avoid potential application crashes. Furthermore, implementing effective testing in the quality assurance process will detect the issue and facilitate its proper resolution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P10 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | CodeSonar version 6.0p0 specializes in detecting and resolving memory leaks in software applications. |
| Helix QAC | 2021.1 | Build-in static code analysis | For Helix QAC version 2021.1, you can use its built-in static code analysis capabilities to check for compliance with coding standards and identify potential issues in your codebase. |
| LDRA tool suite | 9.7.1 | 50 D | LDRA Tool Suite version 9.7.1 focuses on detecting and resolving issues related to rule 50 D to improve code quality and compliance. |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-ERR57-a | Parasoft C/C++test version 2020.2 targets the CERT\_CPP-ERR57-a guideline, likely focusing on detecting and resolving issues related to error handling in C++ code to ensure compliance with established standards and improve software reliability. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | Range check element access |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example that does not adhere to standards, the value returned by the call to get\_index() might exceed the number of elements stored in the string, leading to undefined behavior. |
| #include <string>  extern std::size\_t get\_index();  void f() {  std::string s("01234567");  s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| In this compliant approach, the std::basic\_string::at() function is utilized, which functions similarly to the index operator[], but throws a std::out\_of\_range exception if pos >= size(). |
| #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):** Use Effective Quality Assurance Techniques & Adopt a Secure Coding Standard - Implementing effective quality assurance methods is essential; it serves as a gatekeeper for enforcing proper element checks, as neglecting to validate elements can result in out-of-bounds reads or writes. Embracing a secure coding standard is crucial, as it ensures that elements remain within the appropriate range, safeguarding the program against vulnerabilities stemming from unchecked elements. |
| --- |

**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 | Astrée version 22.10 detects and addresses assert failures, indicating a focus on identifying and resolving issues related to failed assertions in software code for improved reliability and robustness. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | CodeSonar version 7.4p0 targets various memory-related issues including buffer overflows, uninitialized memory accesses, and other memory-related vulnerabilities, aiming to enhance software security and reliability. |
| [Parasoft C/C+ + test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-MEM56-a | Parasoft C/C++ Test version 2023.1 targets the CERT\_CPP-MEM56-a guideline, likely focusing on memory management practices in C++ code to ensure adherence to established standards and improve code robustness. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023A | CERT C++: STR53-CPP | Polyspace Bug Finder version R2023a addresses the CERT C++ STR53-CPP guideline, focusing on detecting and resolving issues related to string handling in C++ code to ensure compliance with established standards and improve software reliability. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | Close open files when not in use and no longer needed |

| **Noncompliant Code** |
| --- |
| This code example is not compliant because the resource allocated by the call to fopen() is not closed before the program terminates. While exit() does close the file, the program lacks a means of determining if an error occurs during the flushing or closing of the file. |
| #include <stdio.h>  #include <stdlib.h>  int main(void) {  FILE \*f = fopen(filename, "w");  if (NULL == f) {  exit(EXIT\_FAILURE);  }    /\* ... \*/  exit(EXIT\_SUCCESS);  } |

| **Compliant Code** |
| --- |
| In this compliant code, the program explicitly closes "f" before invoking exit(), enabling any errors that occur during the flushing or closing of the file to be handled appropriately. |
| #include <stdio.h>  #include <stdlib.h>  int main(void) {  FILE \*f = fopen(filename, "w");  if (NULL == f) {  /\* Handle error \*/  }  /\* ... \*/  if (fclose(f) == EOF) {  /\* Handle error \*/  }  exit(EXIT\_SUCCESS);  } |

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

| **Principles(s):** Adopt a Secure Coding Standard - By properly closing files, the risk of information leakage or the consumption of unnecessary resources in the program can be mitigated. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | Assert\_failure | Astrée version 22.10 detects and addresses assert failures, indicating a focus on identifying and resolving issues related to failed assertions in software code for improved reliability and robustness. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | ALLOC.LEAK | CodeSonar version 7.4p0 specializes in detecting and addressing memory leaks, concentrating on identifying instances where memory allocation occurs but is not properly deallocated, thereby enhancing software reliability and resource management. |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | [Insert text.] | RESOURCE\_LEAK | Coverity, with its designated version, focuses on detecting and resolving resource leaks in code, ensuring proper management of system resources and enhancing software reliability. |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023A | CERT C: Rule F1O42-C | Polyspace Bug Finder version R2023A targets CERT C: Rule F1O42-C, indicating its focus on detecting and addressing issues related to file input/output in C code to ensure compliance with established standards and improve software reliability. |

### 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.

[Insert your written explanations here.]

### 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 | Probable | High | P6 | L2 |
| STD-003-CPP | High | Likely | Medium | P1 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | Medium | Probable | Medium | P4 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Medium | P10 | L3 |
| STD-008-CPP | Low | Probable | High | P10 | L3 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-010-CPP | Medium | Unlikely | 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 | The safeguarding of data stored on devices, known as encryption at rest, refers to the encryption of data that is not actively in use, such as data stored on a hard disk or database, with the objective of safeguarding it in case of a breach or theft. In such a scenario, attackers would need to either employ brute force to decrypt the data, which is time-consuming, or obtain access to the encryption keys. This significantly prolongs the process and reduces the effectiveness of any stolen data. |
| Encryption in flight | Describing encryption in flight, or securing data during transit from point A to point B, highlights the vital practice of safeguarding information as it moves between network and user or vice versa, ensuring protection against interception and unauthorized access. Encrypting data during transit, followed by decryption upon arrival, is crucial in preventing potential breaches and minimizing their impact. |
| Encryption in use | Guaranteeing the encryption of sensitive data throughout all stages is essential. Encryption in use ensures continuous security, aligning with the defense-in-depth approach where multiple layers of defense are deployed at every potential point of attack, ensuring constant protection. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of individuals accessing the system, allowing the system to accept or reject them based on predefined criteria, which is crucial for safeguarding sensitive data. This policy is applicable to all users seeking access, emphasizing the importance of implementing an authentication system as a primary layer of defense. |
| Authorization | Authorization follows authentication and determines the level of access granted to a user for a system, its files, and its resources, typically through role-based permissions defining their actions. Aligned with the principle of Default Deny, this ensures that new users, even when authenticated, lack inherent access to resources until assigned a specific role, preventing unauthorized data access. Moreover, it facilitates seamless elevation of user privileges to higher tiers with enhanced resource access. |
| Accounting | Accounting involves maintaining a record of users' actions within the system, providing administrators with a backlog to review in case of suspicious activities. This approach also facilitates the detection of unauthorized attempts to access or modify the system. |

**\***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 | 03/27/2024 | Initial Template | David Buksbaum |  |
| 1.1 | 03/27/2024 | Changed the requirements for milestone assignment: Included descriptions and codes. | Mark Eilers |  |
| 1.2 | 04/13/2024 | Evaluated the severity of the vulnerabilities | Mark Eilers |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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