**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 | Validating input data stands as a primary defense against malicious attacks in software applications. This approach involves scrutinizing data before its utilization or processing within the system. Swiftly identifying and flagging suspicious data upon entry into the system is crucial to prevent potential vulnerabilities from being exploited. By implementing mechanisms that assess and validate information, both from trusted and untrusted sources, the system's security can be significantly bolstered. Various types of data inputs pose risks, predominantly originating from untrusted external sources. Nonetheless, thorough validation of input data remains instrumental in safeguarding the system. |
| 1. Heed Compiler Warnings | It's crucial for developers to pay attention to warnings, as they provide valuable insights that can prevent bugs and unexpected issues. Neglecting warnings may lead to problems down the line, requiring additional resources to resolve. While warnings themselves may not necessarily indicate unsafe code, failing to address them promptly can escalate into larger issues. These warnings typically surface during the code compilation process and should be prioritized to ensure the smooth operation of the program. Employing dynamic and static analysis methods can help generate compiler warnings, enabling the detection of flaws that may halt the program execution. |
| 1. Architect and Design for Security Policies | When crafting the application architecture, it's vital to integrate security measures from the outset. Failing to address and implement security protocols until later stages can lead to unexpected complications and necessitate significant project restructuring. By integrating security policies throughout each phase of architecture design, a robust and secure project can be developed. |
| 1. Keep It Simple | The level of complexity within the system directly influences its performance, maintainability, and readability. Simplifying the system and incorporating clear comments throughout the code enhances the final outcome and minimizes the likelihood of errors and bugs. |
| 1. Default Deny | Implementing a default deny approach as a security measure within the system holds significant importance and value. Default deny enhances system security by automatically denying access to sensitive components such as the database and user accounts, unless proper authentication credentials are provided. This proactive measure helps safeguard the program against unauthorized access and potential security breaches initiated by users with malicious intent. |
| 1. Adhere to the Principle of Least Privilege | Limiting user access within the system can effectively mitigate exploitation risks and enhance accountability. This approach not only helps mitigate potential risks but also fosters a culture of accountability within the organization. By restricting access to only a select few employees with appropriate credentials, it becomes possible to maintain a comprehensive log of system modifications and user activity. Implementing this strategy proves advantageous in both risk reduction and maintaining an accurate record of system changes. |
| 1. Sanitize Data Sent to Other Systems | Verifying and validating data before transmission to external sources serves as a crucial defense against the dissemination of malicious or compromised data. By confirming the cleanliness of the data and ensuring its absence of malware, the risk of infecting external systems is effectively mitigated. This rigorous process plays a pivotal role in upholding a reputable reputation and safeguarding against potential threats. |
| 1. Practice Defense in Depth | The implementation of multiple security layers to prevent malware from infiltrating the system is referred to as defense in depth. This approach involves creating a series of protective barriers that act as successive lines of defense in the event that one layer is breached. While establishing various security layers is essential for ensuring a robustly secured system, there are important considerations to bear in mind when adopting this strategy. However, managing and integrating diverse security layers can become complex over time due to the unique nature of each layer. |
| 1. Use Effective Quality Assurance Techniques | Employing efficient quality assurance methods leads to enhanced accountability, thorough testing, iterative improvements, and ultimately, the delivery of optimal outcomes. While there are various approaches to implementing this methodology, the fundamental principles for achieving superior results are closely linked to comprehensive testing and iterative enhancements of existing or past work. Evaluating code from both developer and user perspectives enables the acquisition of valuable insights into performance and user experience, facilitating the refinement of the system and culminating in the development of a robust and well-founded project. |
| 1. Adopt a Secure Coding Standard | Integrating secure coding standards is not only beneficial but also essential for every project. The absence of such standards would result in unaddressed vulnerabilities, potentially leading to compromised data and system breaches. Upholding coding consistency, implementing measures for vulnerability prevention, conducting thorough code reviews, and providing ongoing training are among the practices that contribute to the establishment of secure coding standards. While specific coding standards may vary slightly based on individual company preferences, they generally encompass similar principles aimed at enhancing system security. Employing secure coding practices is applicable and should always be prioritized regardless of the development language or project at hand. |

### 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-range enumeration value |

| **Noncompliant Code** |
| --- |
| In this example, there's an attempt to validate whether a provided value falls within the acceptable range of enumeration values. However, a critical mistake in the code lies in assigning the enumeration type before verifying if the value falls 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 revised solution, the validation ensures that the value is representable within the enumeration type before proceeding with the conversion. By limiting the converted value to one that corresponds to a specific enumerator value, the risk of encountering an unspecified value during conversion is mitigated. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  } |

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

| **Principles(s):** To avoid complications or errors in calculations, it's crucial to allocate integers/buffers in a manner that is both straightforward and concise. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Cast integer to enum | Checked partially |
| Code Sonar | 7.4po | Lang.case.coerce | Coercion alters the value |
| Parasoft C/C++test | 2023.1 | Cert\_cpp-int5-\_a | An expression with enum that underlies values corresponding to the enumeration |
| Rulechecker | 22.10 | Cast integer to the enum | Partially checked |

#### Coding Standard 2

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

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, the variable "pos" becomes invalidated following the initial call to insert(). As a result, subsequent iterations of the loop exhibit 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) {  d.insert(pos, items[i] + 41.0);  }  } |

| **Compliant Code** |
| --- |
| In this compliant approach, "pos" is consistently assigned a valid iterator during each insertion, effectively averting any instances of 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):** Architecting and designing with security policies in mind ensures that our program's memory usage is carefully considered, helping to prevent typical errors and enhance security. By strictly adhering to proper access protocols, we minimize vulnerabilities and maintain compliance with security regulations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | Alloc.Uaf | Use after free |
| Astree | 22.10 | Overflow\_upon\_derefrence | [Insert text.] |
| Parasaoft C/C++ test | 2023.2 | CERT\_CPP\_CTR51-a | Do not change container while iterating |
| Polyspace bug finder | R2024a | CERT C++ | Checks for the use of invalid iterator. |

#### Coding Standard 3

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

| **Noncompliant Code** |
| --- |
| In this example of noncompliant code, a std::string object is instantiated using the outcome of a std::getenv() call. Nonetheless, since std::getenv() returns a null pointer upon failure, this code may result in undefined behavior when the environment variable is absent (or encounters another error). |
| #include <cstdlib>  #include <string>  void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant approach, the output of the std::getenv() function undergoes a null check before constructing the std::string object. |
| #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):** Ensuring the validation of input data is critical to mitigate risks such as buffer overflow, null pointers, and unpredictable outcomes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace bug finder | R2023a | CERT C++ STR51 – CPP | Check for string operations on null pointer |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STRS1-a | Avoid the null pointer |
| CodeSonar | 7.4 | Lang.mem.npd | Null pointer deference |
| Astree | 22.10 | Assert\_failure |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-cpp] | Do not store an owned pointer value in unrelated smart pointer. |

| **Noncompliant Code** |
| --- |
| The following example involves two independent smart pointers constructed from a shared underlying pointer value. Upon the destruction of the local automatic variable p2, it deallocates the managed pointer value. Consequently, when the local automatic variable p1 is subsequently destroyed, it attempts to deallocate 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 corrected approach involves two std::shared\_ptr objects that are interlinked via copy construction. Upon the destruction of the local automatic variable p2, the usage count for the shared pointer value is decreased but remains above zero. Subsequently, upon the destruction of the local automatic variable p1, the usage count for the shared pointer value is reduced to zero, prompting the deletion of the managed pointer. Additionally, this compliant solution employs 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):** Validating input data serves to safeguard the system against malware infiltration and verifies the safety of the data for processing. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P17 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Dangling pointer use | [Insert text.] |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ MEM56 | [Insert text.] |
| Parasoft C/C++ test | 2024.1 | CERT\_CPP | Do not store already owned smart pointer |
| Polyspace bug finder | R2024a | CERTC++ mem56 CPP | Checks for the use of already owned pointers. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-cpp] | Properly deallocate resources |

| **Noncompliant Code** |
| --- |
| The provided example leads to undefined behavior. It involves passing the local variable "space" as the argument to the placement new operator. Subsequently, the pointer resulting from that call is passed to ::operator delete(), causing undefined behavior as ::operator delete() tries to deallocate memory not allocated by ::operator new(). |
| #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** |
| --- |
| In the compliant code, the correct action is taken by omitting the call to ::operator delete() instead of explicitly invoking the destructor of s1. This scenario illustrates one of the rare occasions where directly invoking a destructor is justified. |
| #include <iostream>  struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };  void f() { |

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

| **Principles(s):** Keep it Simple – Memory management aligns with keeping things simple by preventing  unnecessary the usage of unnecessary bits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Fee non-heap variable. Double  free Type mismatch Leak |
| LDRA tool suite | 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 |
| Clang | 3.9 | ANalyxer | Checked y clang-tidy, but does not  catch all violations of this rule |
| Polyspace bug finder | R2023a | CERT C++ MEM51-CPP | Invalid deletion of pointer  Invalid free of pointer  Deallocation of previously  deallocated pointer |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| The noncompliant code uses the assert() macro to assert() property concerning a memory-mapped structure  that is essential for 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 solution the assertion only involving a constant expression, a preprocessor conditional  statement may be used, as shown 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):** Utilize effective quality assurance techniques to detect issues at an early stage, thus averting their escalation. Employing static assertions serves as a quality assurance measure, aiding in defect elimination. Additionally, prioritize simplicity in programming; avoid unnecessary complexity and utilize assertions in their simplest forms to uncover latent issues within the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axion | 7.2.0 | CertC-DCL | Checked by clang-tidy |
| CodeSonar | 3.9 | Misc | Users can implement a custom  check that reports uses of the  assert() macro |
| Éclair | 1.2 | CC2.DCL03 | Fully Implemented |
| LDRA tool suite | 9.7.1 | 44 s | Fully Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-cpp] | Handle all exceptions thrown before main() begins |

| **Noncompliant Code** |
| --- |
| The example shows the constructor for S may throw an exception that is not caught when globalS is  constructed during program startup. |
| struct S {  S() noexcept(false);  };  static S globalS; |

| **Compliant Code** |
| --- |
| In the compliant solution, the variable globalS is transformed into a local variable with static storage duration. This adjustment ensures that any exceptions thrown during the object's construction can be caught. This is because the constructor for S will execute the first time the function globalS() is invoked, rather than during program startup. However, implementing this solution necessitates modifying the source code so that any previous instances of globalS are substituted with a function call to globalS(). |
| struct S {  S() noexcept(false);  };  S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** : Architect and Design for Security Policies – With a well thought out design developers can  catch exceptions  Keeping it Simple - Developers accustomed to writing straight-forward and clean code make it simple for  others to understand their objective. This contributes to validating if assertions are working as intended  when used. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Potential throw static initial | Partially checked |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP\_ERR58 | Exceptions shall be raised only  after start-up and before  termination of the program |
| Polyspace bug finder | R2024a | CERT C++: ERR58-CPP | Checks for exceptions raised  during program startup |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-cpp] | Do not use pointer-to-member operators to access nonexistent members |

| **Noncompliant Code** |
| --- |
| In the noncompliant example provided, a pointer-to-member object is retrieved from D::g but is subsequently upcasted to a B::\* type. Although invoking it on an object with the dynamic type D is well-defined, the underlying object's dynamic type is B, leading to undefined behavior. |
| struct B {  virtual ~B() = default;  };  struct D : B {  virtual ~D() = default;  virtual void g() { /\* ... \*/ }  };  void f() {  B \*b = new B;  // ...  void (B::\*gptr)() = static\_cast<void(B::\*)()>(&D::g);  (b->\*gptr)();  delete b;  } |

| **Compliant Code** |
| --- |
| In this compliant resolution, the upcast is eliminated, rendering the original code invalid and highlighting the fundamental issue that B::g() does not exist. This approach assumes that the programmer intended to utilize the appropriate dynamic type for the underlying object. |
| struct B {  virtual ~B() = default;  };  struct D : B {  virtual ~D() = default;  virtual void g() { /\* ... \*/ }  };  void f() {  B \*b = new D; // Corrected the dynamic object type.  }  // ...  void (D::\*gptr)() = &D::g; // Moved static\_cast to the next line.  (static\_cast<D \*>(b)->\*gptr)();  delete b |

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

| **Principles(s):** Simplicity in code brings numerous advantages, including the prevention of access to non-existent members. Designing architecture with security policies in mind involves defining clear guidelines that illuminate the consequences of accessing non-existent members, which can result in unpredictable behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace | R2024a | CERT C++ OOP55-CPP | Checks for pointers to member  accessing non-existing class  members |
| Parasoft |  |  | Runtime detect |
| Parasoft C/C++ | 2024.1 | CERT\_CPP-OOP55-a | A cast shall not convert a pointer  to a function to any other pointer  type, including a pointer to a  function type |
| CodeSonar | 7.4 | LANG.MEM | Uninitialized variable |

#### Coding Standard 9

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

| **Noncompliant Code** |
| --- |
| This noncompliant code example may lead to undefined behavior because the value returned by the get\_index() function could exceed the number of elements stored in the string. |
| #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 employed, offering similar functionality to the index operator[]. However, it differs by throwing a std::out\_of\_range exception if the specified position exceeds the size of the string. |
| #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):** Embracing secure coding standards plays a crucial role in verifying that elements reside within the correct range. Neglecting to check elements for their appropriate range leaves the program susceptible to vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace bug finder | R2023a | CERT C++ STR53-CPP | Array Access out of bounds--  Array access with tainted  index  Pointer dereference with  tainted offset |
| Parasoft | 2024.1 | CERT\_CPP | Guarantee that container indices  are within the valid range |
| Codesonar | 7.4p0 | 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 |
| Asstree | 22.10 | Assert\_failure |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-010-cpp] | Close the files that are no longer needed. |

| **Noncompliant Code** |
| --- |
| The provided code example is considered noncompliant due to the lack of closing the resource allocated by the fopen() function before the program exits. While exit() is used to close the file, the program lacks a mechanism to verify if an error occurs during the flushing or closing process. |
| #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 resolution, the program explicitly closes the file "f" before invoking exit(), ensuring that any potential errors during the flushing or closing of the file are appropriately addressed. |
| #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):** Following a secure coding standard involves closing files to prevent information leakage and to avoid unnecessary resource consumption in the program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported with no checker |
| CodeSonar | 7.4p0 | ALLOC.LEAK | Leaks |
| Coverity | 2017.0 | CERT\_C-FIo | Ensure resources are freed up |
| Polyspace bug finder | R2023A | CERT rule FIO42-C | Checks for leaks of resources. |

### 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.

To streamline the enforcement of our standards, establishing a centralized policy repository becomes essential. This repository becomes integral to our evaluation and strategizing processes, serving as a repository for all organizational policies, including regulatory compliance, release prerequisites, and general operational guidelines. This data forms the foundation for initiating policy automation and enforcement. During the validation and evaluation phase, this repository will be utilized. Automated tools will assess risks, issue alerts, and deliver notifications. Automating compliance enhances our productivity and reduces repetitive tasks. Additionally, we can automate the transition and health check phases. Implementing automated penetration testing aids in minimizing false alarms and streamlining issue resolution.

Taking further steps, we can progress towards fully automating the remainder of the production process. Establishing a system to generate and maintain logs within a database enables us to identify vulnerabilities and thwart potential attacks. This is achieved through methods such as signature verification, ensuring data integrity, and implementing multiple layers of security. To uphold the stability and security of our system, automatic save points can be established. These save points serve as pristine backups, enabling a return to a secure version in the event of an attack or system malfunction.

### 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 | 2 |
| STD-003-CPP | High | Likely | Medium | P18 | 1 |
| STD-004-CPP | High | Likely | Medium | P18 | 1 |
| STD-005-CPP | High | Likely | Medium | P18 | 1 |
| STD-006-CPP | Low | Unlikely | High | 1 | 2 |
| STD-007-CPP | Low | Likely | Low | P9 | 2 |
| STD-008-CPP | High | Probable | High | P6 | 2 |
| STD-009-CPP | High | Unlikely | Medium | P6 | 2 |
| STD-010-CPP | Medium | Unlikely | Medium | 4 | 3 |

### 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 | Encrypting data stored on devices is referred to as encryption at rest. Whether the data resides on hard drives or databases, employing encryption ensures its protection in various states. Integrating encryption at rest into policies serves as a safeguard against potential data breaches. Although hackers may access the data, encryption renders it unreadable and therefore useless to unauthorized parties. Without encryption at rest, data becomes vulnerable to exploitation in the event of a system breach. |
| Encryption in flight | Encryption in flight refers to the process of securing data as it traverses from one point to another. Data is in perpetual motion, flowing between networks and users or vice versa. Given the susceptibility of data interception during transit, encrypting data while it is in motion becomes crucial. This measure ensures that unauthorized individuals cannot view the data during transmission. Data is encrypted before being sent and decrypted upon reaching its destination. Implementing this policy helps mitigate the potential damage resulting from a data breach. |
| Encryption in use | Implementing safety measures is essential to limit access to data being manipulated within software applications exclusively to the intended user. Data is particularly vulnerable when in an active state, yet encrypting data while it resides in the computer's memory effectively thwarts eavesdropping attempts by attackers. This policy is vital in safeguarding data and reducing its susceptibility to exploitation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication involves confirming that the user seeking access to the system possesses the requisite credentials. This process restricts system entry to individuals authorized by the administrator. Various authentication methods, including physical key cards, biometric features, or two-factor authentication, may be employed by systems. This policy is relevant for its security attributes, which entail restricting access solely to authorized personnel. |
| Authorization | Authorization pertains to restricting the actions that users can undertake within the network. By limiting user capabilities within the system, the likelihood of malware or vulnerabilities being introduced is diminished. Adhering to the principle of "less is best" when implementing authorization implies that minimizing user access levels enhances the system's ability to fend off attacks. |
| Accounting | Accounting involves maintaining a log of users' activities within the system. Recording the actions taken by individuals while they are on the network enables administrators to review historical data in cases of suspicious activity detection. Additionally, this method can serve to notify administrators of any unauthorized attempts to access or modify data. |

**\***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 |  |
| 2.0 | 4/12/2024 | Project 1 | Vince Taliaferro |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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