**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 | This security principle is used to verify all inputted data is correct and prevents harmful data from being inputted into the system. This can include checking for buffer overflow and other harmful hacks before they reach the system. |
| 1. Heed Compiler Warnings | Compiler warnings are used to identify potential errors and security issues withing the system. A developer needs to pay attention to these warnings in order to identify possible attacks and detect these issues early on. |
| 1. Architect and Design for Security Policies | The system is designed to have security measures for the start of the program. This allows better resistance to attacks. Measures such as DiD (defense in depth) allows multiple layers or security are a part of the design and helps prevent system vulnerabilities. |
| 1. Keep It Simple | Having simple readable code that does not introduce complicated security measures helps to keep the system manageable and easy to understand for developers. Simple security measures are important because if something needs to be patched or updated, it is a simple fix. |
| 1. Default Deny | Access to certain data is denied and will ask for authorization such as a username and password. This adds a layer of security that prevents unauthorized individuals from accessing certain files. |
| 1. Adhere to the Principle of Least Privilege | Only certain data will be available to certain individuals. This prevents changes and data being accessed to individuals that are not in the scope to need or perform tasks using this data. |
| 1. Sanitize Data Sent to Other Systems | Checking for data manipulation and making sure data is encoded before being sent to different system helps prevent SQL injection and other types of attacks. |
| 1. Practice Defense in Depth | Multiple layers of security are added, if one line of defense is broken there will be others to prevent further attacks. |
| 1. Use Effective Quality Assurance Techniques | These techniques consist of using practices like code reviews and automated testing to identify security issues during code development and testing. |
| 1. Adopt a Secure Coding Standard | Using a developed coding standard allows consistency and approved guidelines throughout the development and throughout the developing team and company. |

### 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 | **Avoid casting to an out-of-range enumeration value**  -validate integer before casting to enumeration value to avoid overflow, errors and system instability. |

| **Noncompliant Code** |
| --- |
| **Noncompliant Code Example (Bounds Checking)**  This noncompliant code does not check for a valid integer range before casting a value to the enumeration type. If the value is outside of the range it will still be cast, this can cause unexpected errors and incorrect values. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| **Compliant Solution (Bounds Checking)**  This compliant code validates the integer before casting it to a enumeration type. This is done by using an if statement to make sure the integer is within the range of the enum type. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  }  **Compliant Solution (Scoped Enumeration)**  Allows a value to be converted into an enum value, this ensures code safety but can cause an invalid input being used. |
| enum class EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

**Compliant Code**

|  |
| --- |
| **Compliant Solution (Fixed Unscoped Enumeration)**  Similar to the previous compliant solution, this compliant solution uses an unscoped enumeration but provides a fixed underlying int type allowing any value from the parameter to be converted to a valid enumeration value. |
| enum EnumType : int {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** Validate Input Data: Validating input data before casting to an enumeration, can prevent overflow, and other potential vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024b | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |
| Parasoft C/C++test | 2024.2 | **CERT\_CPP-INT50-a** | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.37 | V1016 | Tool for detection of unsafe enum usage. |
| CodeSonar | 9.0p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Avoid reading uninitialized memory**  -uninitialized memory can contain unpredictable memory that can lead to corrupt data and security issues. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an  expression to print its value, resulting in undefined behavior. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>  void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques: These techniques consist of using practices like code reviews and automated testing to identify security issues during code development and testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2025.1 | **UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST UNINIT.STACK.ARRAY.MIGHT UNINIT.STACK.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST UNINIT.STACK.MIGHT UNINIT.STACK.MUST** | Fully implemented |
| CodeSonar | 9.0p0 | **LANG.MEM.UVAR** | Uninitialized variable |
| Parasoft C/C++test | 2024.2 | **CERT\_C-EXP33-a** | Avoid use before initialization |
| PVS-Studio | 7.37 | [**V573**](https://pvs-studio.com/en/docs/warnings/v573/), [**V614**](https://pvs-studio.com/en/docs/warnings/v614/), [**V670**](https://pvs-studio.com/en/docs/warnings/v670/), [**V679**](https://pvs-studio.com/en/docs/warnings/v679/), [**V1050**](https://pvs-studio.com/en/docs/warnings/v1050/) | Tool for detection of unsafe enum usage. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensure string storage has sufficient space  -Making sure a string has enough space to hold data prevents buffer overflow |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>  void f() {  char buf[12];  std::cin >> buf;  } |

**Noncompliant Code**

|  |
| --- |
| To solve this problem, it may be tempting to use the std::ios\_base::width() method, but  there still is a trap, as shown in this noncompliant code example. |
| #include <iostream>  void f() {  char bufOne[12];  char bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo;  } |

| **Compliant Code** |
| --- |
| Instead of inserting a null terminator, it constructs the std::string object based on the number of characters read from the input stream. |
| #include <fstream>  #include <string>  void f(std::istream &in) {  char buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }  std::string str(buffer, in.gcount());  // ...  } |

**Compliant Code**

|  |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #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: Validating input data before casting to an enumeration, can prevent overflow, and other potential vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | **CERT\_C-STR31-a CERT\_C-STR31-b CERT\_C-STR31-c CERT\_C-STR31-d CERT\_C-STR31-e** | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| Klocwork | 2024.4 | **SV.FMT\_STR.BAD\_SCAN\_FORMAT SV.UNBOUND\_STRING\_INPUT.FUNC** | Fully implemented |
| CodeSonar | 9.0p0 | **LANG.MEM.BO LANG.MEM.TO MISC.MEM.NTERM BADFUNC.BO.\*** | Buffer overrun Type overrun No space for null terminator A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Helix QAC | 2024.4 | **C2840,  C5009, C5038**  **C++0145, C++5009, C++5038**  **DF2840, DF2841, DF2842, DF2843, DF2845, DF2846, DF2847, DF2848, DF2930, DF2931, DF2932, DF2933, DF2935, DF2936, DF2937, DF2938** | Includes memory safety and string overflow rules. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Avoid reading uninitialized memory  -uninitialized memory can contain data that can lead to SQL injection. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

**Noncompliant Code**

|  |
| --- |
| In this noncompliant code example, an int \* object is allocated by a *new-expression*, but the memory it points to is not initialized. |
| #include <iostream>  void f() {  int \*i = new int;  std::cout << i << ", " << \*i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>  void f() {  int i = 0;  std::cout << i;  } |

**Compliant Code**

|  |
| --- |
| In this compliant solution, the memory is direct-initialized to the value 12 prior to printing its value. |
| #include <iostream>  void f() {  int \*i = new int(12);  std::cout << i << ", " << \*i;  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems: Checking for data manipulation and making sure data is encoded before being sent to different system helps prevent SQL injection and other types of attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | **CERT\_C-STR31-a CERT\_C-STR31-b CERT\_C-STR31-c CERT\_C-STR31-d CERT\_C-STR31-e** | Avoid use before initialization |
| CPPcheck | 2.15 | **uninitvar uninitdata uninitstring uninitMemberVar uninitStructMember** | Open source tool, offering security detection |
| Klocwork | 2024.4 | **UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST UNINIT.STACK.ARRAY.MIGHT UNINIT.STACK.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST UNINIT.STACK.MIGHT UNINIT.STACK.MUST** | Fully implemented |
| Helix QAC | 2024.4 | **DF2726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF2978** | Fully implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Avoid accessing freed memory |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

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

**Compliant Code**

|  |
| --- |
| use automatic storage duration instead of dynamic storage duration. Since s is not required to live beyond the scope of g(), this compliant solution uses automatic storage duration to limit the lifetime of s to the scope of g(). |
| struct S {  void f();  };  void g() {  S s;  // ...  s.f();  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques: These techniques consist of using practices like code reviews and automated testing to identify security issues during code development and testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | **USE\_AFTER\_FREE** | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Klocwork | 2024.4 | **UFM.DEREF.MIGHT UFM.DEREF.MUST UFM.FFM.MIGHT UFM.FFM.MUST UFM.RETURN.MIGHT UFM.RETURN.MUST UFM.USE.MIGHT UFM.USE.MUST** | Fully implemented |
| Parasoft C/C++ test | 2024.2 | **CERT\_C-MEM30-a** | Do not use resources that have been freed |
| Helix QAC | 2024.4 | **DF4866, DF4867, DF4868, DF4871, DF4872, DF4873**  **C++3339, C++4303, C++4304** | Fully implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Avoid abruptly terminating the program**  **-using assertions will cause the system to terminate this can stop proper recovery and error reporting.** |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(), may result in a call to std::terminate() because throwing\_func() may throw an exception. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f() handles all exceptions thrown by throwing\_func() and does not rethrow. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):** Keep It Simple: Having simple readable code that does not introduce complicated security measures helps to keep the system manageable and easy to understand for developers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024b | [CERT C++: ERR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr50cpp.html) | Checks for implicit call to terminate() function (rule partially covered) |
| RuleChecker | 22.10 | **stdlib-use** | Partially checked |
| CodeSonar | 9.0p0 | **BADFUNC.ABORT BADFUNC.EXIT** | Use of abort Use of exit |
| Astree | 22.10 | **stdlib-use** | Partially checked |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful 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):** Architect and Design for Security Policies: The system is designed to have security measures for the start of the program. This allows better resistance to attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool Suite | 9.7.1 | 527 S | Partially implemented |
| RuleChecker | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| CodeSonar | 9.0p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointers | STD-008-CPP | Use Valid References, pointers and iterators  -using invalid pointers, references and iterators can lead to data corruptions and system errors |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, pos is invalidated after the first call to insert(), and subsequent loop iterations have 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 solution, pos is assigned a valid iterator on each insertion, preventing 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):** Adopt a Secure Coding Standard: Using a developed coding standard allows consistency and approved guidelines throughout the development and throughout the developing team and company. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | **ALLOC.UAF** | Use after free |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| Polyspace Bug Finder | R2024b | [CERT C++: CTR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr51cpp.html) | Checks for use of invalid iterator (rule partially covered). |
| Klocwork | 2025.1 | **ITER.CONTAINER.MODIFIED** | Fully implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | STD-009-CPP | **Close files when they are no longer needed**  -leaving files open can lead to system memory being used, causing unexpected crashed and performance issues. |

| **Noncompliant Code** |
| --- |
| the underlying std::basic\_filebuf<T> object maintained by the object is not 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;  }  // ...  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):** Adhere to the Principle of Least Privilege: Only certain data will be available to certain individuals. This prevents changes and data being accessed to individuals that are not in the scope to need or perform tasks using this data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2024b | CERT C++:FIO51-CPP | Checks for resource leak (rule partially covered) |
| Helix QAC | 2025.1 | **DF4786, DF4787, DF4788** | Fully implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object oriented Programming | [STD-010-CPP] | Avoid using pointers to access nonexistent members  -using pointers to access deleted or nonexistent data can lead to security vulnerabilities |

| **Noncompliant Code** |
| --- |
| When called on an object whose dynamic type is D, the pointer-to-member  call is well defined. However, the dynamic type of the underlying object is B, which results in  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** |
| --- |
| the upcast is removed, rendering the initial code ill-formed and emphasizing the underlying problem that B::g() does not exist. |
| 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):** Validate Input Data: Validating input data before casting to an enumeration, can prevent overflow, and other potential vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | **CERT\_CPP-OOP55-a** | A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type |
| Polyspace Bug Finder | R2024b | [CERT C++: OOP55-CPP](https://www.mathworks.com/help/bugfinder/ref/certcoop55cpp.html) | Checks for pointers to member accessing non-existent class members (rule fully covered). |
| CodeSonar | 9.0p0 | **LANG.MEM.UVAR** | Uninitialized Variable |
| Helix QAC | 2025.1 | **DF2810, DF2811, DF2812, DF2813, DF2814** | Fully implemented |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

**The existing DevOps process focused on It operations, Application delivery and Development but is lacking in its security. Transferring to a DevSecOps means creating a strong security system that is balanced with the other processes. In the pre-production phase runtime protection and security automation can be introduced. Static code analysis tools such as Parasoft C/C++test can check for system vulnerabilities and errors pre-production. In production tools can be used to check for runtime errors and can protect against issues like buffer overflow and SQL injection attacks from hackers.**

### 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 | 3 |
| STD-002-CPP | High | Probable | Medium | P12 | 1 |
| STD-003-CPP | High | Likely | Medium | P18 | 1 |
| STD-004-CPP | High | Probable | Medium | P12 | 1 |
| STD-005-CPP | High | Likely | Medium | P18 | 1 |
| STD-006-CPP | Low | Probable | Medium | P4 | 3 |
| STD-007-CPP | Low | Probable | Medium | P4 | 3 |
| STD-008-CPP | High | Probable | High | P6 | 2 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | 3 |
| STD-010-CPP | High | Probable | High | P6 | 2 |

### 

### 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 | This protects data stored on physical devices like, hard drives, databases and other system backups. Data cannot be stolen and is protected from security breaches |
| Encryption in flight | Protects data as it moves from one system to another. This protects data such as user logins and communication from being intercepted. |
| Encryption in use | Protects data while being used in a program, this is used because data may become vulnerable when loaded into memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Used to verify user identity. User identification like can protect against unauthorized users and identity hacks |
| Authorization | Provides a user with specific access based on their authorization levels. This prevents users from accessing sensitize data |
| Accounting | Keeps track of what a user does within the system. Firewalls and system logs keeps tracks of user activity, this can look for hacking and help with system audits. |

**\***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 | 6/1/2025 | Updated Template | Shanine Efferson | Philomena Ogoh |
| 3.0 | 6/22/2025 | Completed Template | Shanine Efferson | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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