**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 helps ensure that data input into systems is correct, safe, and helps prevent different types of attacks, like buffer overflows. Ensuring that input data is not longer than the smallest buffer that it is stored in helps ensure that buffer overflow attacks do not occur. |
| 1. Heed Compiler Warnings | Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software. |
| 1. Architect and Design for Security Policies | This is the concept of integrating security practices into the architectural and design phases of the SDLC. This helps ensure that the software will be following security best practices, will prevent threats and attacks, follow regulations, etc. Security practices should be reviewed in these steps to ensure that the software is secure. |
| 1. Keep It Simple | While software security is important and should be considered in depth, keeping it simple is also important as it prevents complexity in many areas. Complex security measures make it more difficult to implement and leave room for more errors and bugs, keeping it simple ensures that it is done correctly and effectively, maintenance is easier when it is kept simple, user experience will not experience as many delays or lag as a complex security may create, it is more cost effective, etc. |
| 1. Default Deny | This is the practice of denying access default unless explicitly given permission. This helps prevent unauthorized access, enhances security, etc. |
| 1. Adhere to the Principle of Least Privilege | This is the practice of only allowing the minimum amount of access to users, processes, and systems as necessary. Limiting access to the bare minimum helps prevent threats from the inside, enhances security, etc. |
| 1. Sanitize Data Sent to Other Systems | Data should be validated and cleansed before being sent to any other external systems, this helps prevent injection attacks, protects sensitive data, ensures data integrity, etc. Data sanitization also refers to the permanent deletion of sensitive data so that it can not be recovered, this helps prevent attacks and recovery of sensitive data by attackers. |
| 1. Practice Defense in Depth | Defense in depth (DiD) is the practice of creating multiple security layers that protect against a multitude of attacks. A singular security measure will not prevent all possible attacks, so multiple security measures should be taken. This helps eliminate redundancy in security, if one layer fails there is another, follows best practices, etc. |
| 1. Use Effective Quality Assurance Techniques | This helps ensure that the software is secure and prevents software attacks. Vulnerabilities should be identified in the QA phase, plans to correct vulnerabilities should be made, and so on. Ensuring that effective QA is in place helps ensure that all vulnerabilities are found through effective testing and attempts to break through the security. |
| 1. Adopt a Secure Coding Standard | This is the practice of adopting guidelines, rules, standards, and best practices. This ensures that the code being written is following best practices to ensure that security vulnerabilities are being protected against. |

### 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. Doing so can create unspecified behaviors and result in errors. |

| **Noncompliant Code** |
| --- |
| In this example, the code attempts to check if a value is within range after casting to the enumeration type. This results in errors. |
| 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 example the code checks that the value can be represented by the enumeration type before casting to the enumeration type. This ensures that the conversion will not result in unspecified values. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):**  Heed Compiler Warnings – This helps with early detection of bugs; in this case it would help discover that checking if the value could be represented by the enumeration type BEFORE casting will prevent errors and bugs.  Keep it simple – Complex designs make implementation more difficult and make it harder to find the bugs.  Use effective quality assurance techniques – This ensures that effective plans to correct errors and bugs will be put in place.  Adopt a secure coding standard – Following rules, guidelines, standard, and best practices help prevent errors, even this enumeration type error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration. |
| RuleChecker | 22.10 | Cast-integer-to-enum | Partially checked. |
| CodeSonar | 8.1p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion alters value.  Cast alters value. |
| Polyspace Bug Finder | R2024a | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not qualify a reference type with const or volatile. Attempting to cv-qualify a reference can result in undefined behavior and errors. |

| **Noncompliant Code** |
| --- |
| In this example a const-qualified reference to a char is created instead of referred to. |
| #include <iostream>    void f(char c) {  char &const p = c;  p = 'p'; // Error: read-only variable is not assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| Removing the const qualifier prevents this error from occurring. |
| #include <iostream>    void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):**  Heed compiler warnings – checking the warnings helps find where the errors are started and how to fix them.  Keep it simple – keep the software simple, easy to follow, and to the point. Using const when it is not needed leads to bugs and could easily be avoided.  Use effective quality assurance techniques – This is an error that could easily be found in the QA stage, ensuring that the QA techniques are effective is essential to finding and preventing errors and bugs.  Adopt a secure coding standard – creating rules, standards, guidelines, and following best practices can help acoid this const/volatile error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile. |
| Polyspace Bug Finder | R2024a | CERT C++: DCL52-CPP | Checks for const-qualified reference types and modification of const-qualified reference types. |
| Clang | 3.9 |  | Produces an error when this standard is violated without the need to specify any special flags or options. |
| Klocwork | 2024.1 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer as it can lead to undefined behaviors and errors. |

| **Noncompliant Code** |
| --- |
| In this example, a std::string object is created from a function that returns a null pointer if it fails. This can creat undefined behavior and errors in your software. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this example the function call is checked to ensure it is not null before creating 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):**  Heed compiler warnings – reading the compiler warnings helps find bugs and errors and prevent them.  Keep it simple – keep the code simple and easier to follow, this makes it easier to find the problems.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Creating a coding standard helps us to prevent these kind of errors as well, for instance creating/following a standard that tells us to always check our values before using them could prevent errors and bugs in the future. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing. |
| Polyspace Bug Finder | R2024a | CERT C++: STR51-CPP | Checks for string operations on null pointers. |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null pointer dereference. |
| Astrée | 22.10 | assert\_failure |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not write syntactically ambiguous declarations. Code can be misunderstood and can result in errors. |

| **Noncompliant Code** |
| --- |
| In this example an anonymous local variable is expected to local and unlock the mutex m, but the declaration in syntactically ambiguous and can be interpreted as declaring an anonymous object m and default constructing it. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex>(m);  shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this example the lock object is given a different identifier and the correct constructor is called. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex> lock(m);  shared\_resource += 42;  } |

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

| **Principles(s):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – Keep it simple to prevent confusion, faster detection of bugs leads to a faster resolution.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Creating a coding standard helps us to prevent  Adopt a secure coding standard – Follow standards and best practices to help prevent these types of errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **P2** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.FNEST | Nested function declaration. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b  CERT\_CPP-DCL53-c | Parameter names in function declarations should not be enclosed in parentheses.  Local variable names in variable declarations should not be enclosed in parentheses.  Avoid function declarations that are syntactically ambiguous. |
| LDRA tool suite | 9.7.1 | 296 S | Partially implemented |
| Polyspace Bug Finder | R2024a | CERT C++: DCL53-CPP | Checks for declarations that can be confused between function and object declaration, and unnamed object or function parameter declaration. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Detect and handle memory allocation errors. It is important to check that the return value of a memory allocation is not nullptr before accessing the resulting pointer. |

| **Noncompliant Code** |
| --- |
| In this example the results of allocation are not checked and the function is labeled noexcept, causing the compiler to assume that there will be no exceptions thrown. This can lead to abnormal termination of the program. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| In this example the returned pointer is checked to ensure it is not nullptr before referencing it and handles the error condition. |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // Handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

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

| **Principles(s):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – Keeping it simple helps ensure there is no confusion.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Follow standards and best practices to avoid these errors, for instance always check variables before assuming they won’t throw an exception, that way errors can be handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled. |
| Polyspace Bug Finder | R2024a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Checks return value of new.  DO not allocate resources in function argument list because the order of evaluation of a function’s parameters in undefined. |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use static assertion to test the value of a constant expression. Doing so will ensure that if a constant-expression us true, nothing will happen, yet if it is false an error message will be output at compile time. |

| **Noncompliant Code** |
| --- |
| This example uses the assert() macro in runtime. It only executes if the code path containing the assertion is executed, if an error occurs before it will not be executed. |
| #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** |
| --- |
| This example uses static\_assert, which allows the assertion to be diagnosed at compile time. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) +  sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – keep the software simple, easy to follow, and to the point.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Follow secure coding standards to avoid and prevent errors, in this case using static\_assert() instead of assert() leads to more effective code and eliminates the error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions. This prevents abnormal terminations and controls termination. |

| **Noncompliant Code** |
| --- |
| In this example, the f() function and main() do not catch exceptions, resulting in abnormal termination. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this example all exceptions are handled. main() uses a try catch block to ensure that exceptions are handled. |
| 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):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the over quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – Keep it simple but effective to ensure there are no bugs or confusion.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – In this case, following a standard that tells you to handle all exceptions could eliminate the possibility of this error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable catch. |
| Polyspace Bug Finder | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions.  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point. |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-008-CPP] | Ensure that integer conversion do not result in lost or misinterpreted data. |

| **Noncompliant Code** |
| --- |
| In this example, from unsigned to signed, results in an error. |
| #include <limits.h>    void func(void) {  unsigned long int u\_a = ULONG\_MAX;  signed char sc;  sc = (signed char)u\_a; /\* Cast eliminates warning \*/  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| In this example there is an error handler and the conversion is correct. |
| #include <limits.h>    void func(void) {  unsigned long int u\_a = ULONG\_MAX;  signed char sc;  if (u\_a <= SCHAR\_MAX) {  sc = (signed char)u\_a; /\* Cast eliminates warning \*/  } else {  /\* Handle error \*/  }  } |

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

| **Principles(s):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – Keep it simple and effective to avoid bugs and errors.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Use error handlers and check variables in the case that there is lost or misinterpreted data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char. |
| TrustInSoft Analyzer | 1.38 | signed\_downcast | Verified. |
| Polyspace Bug Finder | R2024a | CERT C: Rule INT31-C | Checks for integer conversion overflow, call to memset with unintended value, sign change integer conversion overflow, tainted sign change conversion, and unsigned integer conversion overflow. |
| LDRA tool suite | 9.7.1 | 94 S, 433 S, 434 S | Partially implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Termination | [STD-009-CPP] | Do not abruptly terminate the program. This can result in exceptions and errors. |

| **Noncompliant Code** |
| --- |
| In this example, the call to f() may result in a call to std::terminate(), since there is an exit handler registered with std::at\_exit() that 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 example, f() handles all exceptions and prevents the std::terminate() call. |
| #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):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – Keep it simple and effective to avoid bugs and errors.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Abruptly terminating can cause more errors and exceptions, avoid it to prevent this. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort.  User of exit. |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced enforcement. |
| Polyspace Bug Finder | R2024a | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function. |
| RuleChecker | 22.10 | stdlib-use | Partially checked. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| STD Namespaces | [STD-010-CPP] | Do not modify the standard namespaces. This can result in conflicting identifiers which could result in errors. |

| **Noncompliant Code** |
| --- |
| In this example, x is added to the namespace std resulting in undefined behavior. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| In this example, x is added to nonstd namespace instead of std to prevent conflicting identifiers and errors. |
| namespace nonstd {  int x;  } |

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

| **Principles(s):**  Heed compiler warnings – Paying attention to compiler warnings helps identify possible security vulnerabilities, early detection of bugs, enhances the overall quality of the code, and helps ensure that you are following best practices in your software.  Keep it simple – Keep it simple and effective to avoid bugs and errors.  Use effective quality assurance techniques – This is another error that could be easily found in the QA phase, ensuring that QA techniques are effective helps us to prevent these kind of errors.  Adopt a secure coding standard – Ensuring that we do not use reserved names or modify standard namespaces helps us avoid this type of error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.SNM | Modification of standard namespaces. |
| Helix QAC | 2024.1 | C++3180, C++3181, C++3182 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces ‘std’ and ‘posix’. |
| Polyspace Bug Finder | R2024a | CERT C++: DCL58-CPP | Checks for modification of standard namespaces |

### 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.

Automation should be used throughout the entire process and after release. This helps enhance efficiency, find and correct errors, provide a better experience, etc. During the design and build of the product, automation helps find errors and bugs faster, and leads to a faster resolution of those problems. After release, it is assumed that the product will continue to be updated and improved, automation can help ensure that these updates and improvements do not introduce new bugs or errors.

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

### 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 | Encryption at rest is the practice of encrypting data when it is stored on a physical drive (when the data is at “rest”), like a hard drive, SSD, etc. This ensures that the data remains confidential and cannot be accessed by unauthorized users. Encryption at rest should be used when securing sensitive data that could be used for malicious purposes if accessed by unauthorized users. |
| Encryption in flight | Encryption in flight is the practice of encrypting data while it is being transmitted from one location to another (while it is in “flight”), this includes private or wireless networks. This helps prevent attacks like man-in-the-middle attacks, where the data could be intercepted. Securing communications helps ensure that this does not happen, and if intercepted it still cannot be accessed. |
| Encryption in use | Encryption is use is the practice of protecting and encrypting data while it is being processed. Allowing data to be utilized without needing to decrypt it first helps prevent data breaches. Ensuring that data in memory stays encrypted helps prevent users with malicious intents from gaining access to it. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the practice of verifying the identity of a user before granting access to data. This helps prevent unauthorized access and protects sensitive data. Using roles, passwords, MFA, etc. helps ensure that authentication is in place.  User logins should be used to authenticate users and protect data.  Addition of new users should also be authenticated. |
| Authorization | Authorization is the practice of assigning roles and permissions to particular authenticated users. Assigning roles is one of the main ways this takes place, for instance admin roles may be able to access more data than user roles. This helps prevent data breaches.  Addition of new users should not begin with permissions allowing them to access sensitive data, an authorized user should only add these permissions and roles to a user if they are trusted.  User level of access relates to the roles I discussed above, not all users should have the same level of access to data to help prevent breaches and leaks of data.  Not all files should be visible or accessed by all users. |
| Accounting | Accounting is the practice of logging and monitoring user activities and system events. This helps detect potential security breaches or attacks during and after they occur. This also helps identify the source or cause of the breach or attack.  Changes to a database should be accounted for and logged to ensure that no malicious activities are taking place.  Addition of new users should also be accounted for as well as their permissions, roles, etc.  Files accessed by users should be accounted for in case they relate to a breach or attack, this could help prevent and identify the source of breaches, attacks, or leaks. |

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.11 | 05/26/2024 | Milestone | Jessie Smith | [Insert text.] |
| 1.12 | 06/14/2024 | Project One | Jessie Smith | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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