**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 4](#_Toc52464058)

[Coding Standard 1 5](#_Toc52464059)

[Coding Standard 2 7](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 20](#_Toc52464067)

[Coding Standard 10 21](#_Toc52464068)

[Defense-in-Depth Illustration 23](#_Toc52464069)

[Project One 23](#_Toc52464070)

[1. Revise the C/C++ Standards 23](#_Toc52464071)

[2. Risk Assessment 23](#_Toc52464072)

[3. Automated Detection 23](#_Toc52464073)

[4. Automation 23](#_Toc52464074)

[5. Summary of Risk Assessments 24](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 24](#_Toc52464076)

[7. Map the Principles 25](#_Toc52464077)

[Audit Controls and Management 26](#_Toc52464078)

[Enforcement 26](#_Toc52464079)

[Exceptions Process 26](#_Toc52464080)

[Distribution 27](#_Toc52464081)

[Policy Change Control 27](#_Toc52464082)

[Policy Version History 27](#_Toc52464083)

[Appendix A Lookups 27](#_Toc52464084)

[Approved C/C++ Language Acronyms 27](#_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 | Any input coming from an external source should be treated as untrusted, so input validation should be applied into source code any time data is entered from outside of the program. Validating input prevents malicious input from entering the program and exploiting vulnerable code or causing undefined behavior. Putting this into practice can prevent exploits like SQL injection, buffer overflow, or entering unexpected data types. An example is ensuring buffer overflow does not occur when reading input into an array. This can be done with methods such as range checking, whitelisting characters, checking for correct data type, etc. |
| 1. Heed Compiler Warnings | Paying attention to compiler warnings can alert the developer to potential bugs that may otherwise be difficult to find. The code might be technically correct and still run but a warning can inform of possible unintended behavior. Any time there is a warning the developer should look at it as an error and address it to avoid issues down the line. For example, addressing a possible loss of data warning can prevent losing data when converting from a larger type to a smaller. This principle should be applied always when working in an IDE by using the highest possible warning level and refactoring code to eliminate all warnings. |
| 1. Architect and Design for Security Policies | This principle should be applied when developing a system that may contain security risks. This principle determines that the overall architecture and design of the program should be based around the outlined security policies. It should be done when designing the system structure by reviewing and including all security policies and carried out when writing the source code. Examples may be correctly structuring classes and methods, using OOP principles, or dividing a system into subsystems for different privileges. |
| 1. Keep It Simple | A program should be as simple as possible. Unnecessarily complex systems are harder to maintain, are less readable, and it is more difficult to find and fix bugs within them. For instance, in a large and complex program it is more difficult to implement and ensure security measures due to the system’s complicated nature. This principle should be practiced whenever a program is being written in an IDE, especially when it will be used in the real world, and can be accomplished by writing clean, structured code and keeping intricacies to a minimum. |
| 1. Default Deny | Access should be denied by default and will only be granted by meeting the appropriate conditions. This should be done when designing a system with different user types wherever there are areas of the program that should only be accessed by certain users, such as a company database or an online learning platform. This can be accomplished by basing access on receiving permission instead of on excluding functionality that isn’t allowed. |
| 1. Adhere to the Principle of Least Privilege | Processes should be carried out only with the minimum number of resources and privileges required to complete the task, and any special permissions should only be granted for the least amount of time possible. This helps prevent malicious users from abusing any elevated privileges. This is done when designing and writing the program whenever elevated privileges are needed and can be done by granting permission as needed and taking it away when the process has been executed. |
| 1. Sanitize Data Sent to Other Systems | Make sure that the data being passed adheres to a complex subsystem’s requirements because special characters may invoke unwanted actions. Not doing so can make the system vulnerable to injection attacks. This principle should be carried out whenever data is being output to another subsystem by methods such as eliminating undesirable characters or ensuring that the data is of the correct type. An example is when inputting data to a buffer and using that as an argument in a system() call. |
| 1. Practice Defense in Depth | Use multiple overlapping and redundant layers of defense when securing a system so that if one layer cannot defend against an attack, another layer will be able to prevent it. An example is using multiple security measures when protecting a computer like a firewall, an antivirus, using updated software, and using a network intrusion protection system. This principle should be followed when setting up or designing a system and can be accomplished by using multiple secure coding practices and using the appropriate strategies for the specific type of data being protected. |
| 1. Use Effective Quality Assurance Techniques | Using quality assurance techniques that are standard and effective will help to prevent or identify vulnerabilities in the program. This should be done during planning, development, and even after deployment and is done by adopting techniques that are known and accepted to be useful. Examples of QA techniques are penetration testing, source code audits, or benchmarking. |
| 1. Adopt a Secure Coding Standard | Use a secure coding standard that works for the specific language, platform, and program that is being developed. Doing so will help prevent, detect, or eliminate software vulnerabilities within the program and should be done whenever a system is being developed by using a standard that fits specifications. An example is the SEI CERT C++ coding standard that lays out many rules and recommendations for secure coding. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Do Not Delete an Array Through a Pointer of the Incorrect Type** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | To delete an object with a static type different from its dynamic type, the static type must be a base class of the dynamic type and the static type must have a virtual destructor. Deleting an array through a pointer of the incorrect type will cause undefined behavior, even if the previous conditions are met. |

| **Noncompliant Code** |
| --- |
| An array of Derived objects is created with the pointer of type Base \*, and then is deleted. This will result in undefined behavior, even though the Base destructor is declared as virtual. |
| struct Base {  virtual ~Base(){}  };  struct Derived : Base {};  void f() {  Base \*b = new Derived[10];  delete [] b;  } |

| **Compliant Code** |
| --- |
| Here the pointer is of type Derived \* and removes undefined behavior. |
| struct Base {  virtual ~Base(){}  };  struct Derived : Base {};  void f() {  Derived \*b = new Derived[10];  delete [] b;  } |

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

| **Principles(s):**  *3. Architect and Design for Security Policies* – This standard maps to this principle because creating a design with security policies in mind will help to avoid this particular issue in the first place.  *10. Adopt a Secure Coding Standard* – This standard maps to this principle because defining and adopting a secure standard will but practices in place that will help prevent this vulnerability from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppDepend | v2023.1 | “Do not access a variable through a pointer of an incompatible type”  https://www.cppdepend.com/cert-c |  |
| klocwork | 2020.1 | “CERT.EXPR.DELETE\_ARR.BASE\_PTR”  https://help.klocwork.com/current/en-us/concepts/certcommunitycandccheckerreference.htm | Good for large codebases and custom checkers, covers security |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Value-Returning Functions Must Return a Value from All Exit Paths** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | If a function is expected to return a value, it must return a value from all code paths. If a function takes a path that does not return it will cause undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this example there is no return value for if the character is not a digit. When main()calls isNumber(‘f’), the behavior is undefined because the code path did not return and number does not hold a valid value. |
| bool isNumber(char a) {  if (std::isdigit(a)) {  return true;  }  }  int main () {  bool number = isNumber(‘f’);  std::cout << number << std::endl;  } |

| **Compliant Code** |
| --- |
| Here the function now returns something for all code paths, removing undefined behavior. |
| bool isNumber(char a) {  if (std::isdigit(a)) {  return true;  }  return false;  }  int main () {  bool number = isNumber(‘f’);  std::cout << number << std::endl;  } |

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

| **Principles(s):**  *2. Heed Compiler Warnings –* This standard maps to this principle because the compiler will warn of this problem.  *3. Architect and Design for Security Policies* – This standard maps to this principle because designing a system with security in mind will help avoid mistakes like these.  *4. Keep it Simple* – This standard maps to this principle because overcomplicated code, such as having too many conditional statements or exit paths can cause a developer to overlook a path. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | “Missing ‘return’ in non-void function”  https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/ | Open-source, cross-platform, free |
| CppDepend | V2023.1 | “No return statement in non-void function causes undefined behavior”  https://www.cppdepend.com/cppdependrules.html |  |
| PVS Studio | 7.16 | “Non-void function must return a value”  https://pvs-studio.com/en/pvs-studio/sast/cert/ |  |
| klocwork | 2020.1 | “FUNCRET.GEN”  https://help.klocwork.com/current/en-us/concepts/candccheckerreference.htm | Good for large codebases and custom checkers, covers security |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that Storage for Strings has Sufficient Space for Character Data and the Null Terminator** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | For C-style strings, a null character is used to indicate the end of a string. If input is unbounded then the code could lead to buffer overflow and cause undefined behavior. |

| **Noncompliant Code** |
| --- |
| This function does not guarantee enough storage space for the input. If a user enters too many characters then it could result in buffer overflow. |
| void getInput() {  char input[20];  std::cin >> input;  } |

| **Compliant Code** |
| --- |
| Here the function uses getline() with a limit of 20, which causes truncation of the input if it is longer than the limit. This guarantees enough space in the buffer. A call to cin.clear() is needed to clear potential error flags. |
| void getInput() {  char input[20];  std::cin.getline(input, 20);  std::cin.clear();  } |

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

| **Principles(s):**  *1. Validate Input* – This standard maps to this principle because validating input length is one method of avoiding this issue.  *8. Practice Defense in Depth* – This standard maps to this principle because preventing an error like this adds a layer of defense.  *10. Adopt a Secure Coding Standard* – This standard maps to this principle because adopting a secure standard can outline ways to accept input and will likely avoid reading directly into an array of fixed size. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | “Partial string write that leads to buffer that is not zero terminated”  https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/ | Open-source, cross-platform, free |
| Clang-Tidy | 17.0.0git | bugprone-not-null-terminated-result  https://clang.llvm.org/extra/clang-tidy/checks/bugprone/not-null-terminated-result.html | Clang-based C++ linter tool |
| klocwork | 2020.1 | “NNTS.MIGHT”  https://help.klocwork.com/current/en-us/concepts/candccheckerreference.htm | Good for large codebases and custom checkers, covers security |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize Data Passed to Complex Subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Sanitize all string data passed to complex subsystems because special characters can trigger unwanted commands or actions. |

| **Noncompliant Code** |
| --- |
| This code segment gets an email address into a buffer and uses it as an argument in a call to system(). Since the data is not sanitized, the user can enter a malicious string to execute harmful code. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| A whitelist can sometimes be used to sanitize data. A list of acceptable characters is provided so that the program can remove unwanted characters. |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data;  const char \*end = user\_data + strlen(user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):**  *3. Architect and Design for Security Policies* – This standard maps to this principle because while architecting and designing a system, it will be clear when data will need to be sent to another system. This will help developers know when data needs to be sanitized before being sent.  *7. Sanitize Data Sent to Other Systems* – This standard maps to this principle because they are essentially referring to the same problem.  *10. Adopt a Secure Coding Standard* – This standard maps to this principle because a secure coding standard should cover issues such as these and help prevent them from happening. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| klocwork | 2022.4 | Custom | This check may be possible by utilizing klocwork’s ability to create custom checkers |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do Not Access Freed Memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Accessing a pointer that has been deallocated results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this example myStruct is deallocated with the delete operator, and then it is dereferenced to access the struct’s function. |
| struct MyStruct {  void doSomething(){}  }  void myFunction() {  MyStruct \*myStruct = new MyStruct;  delete myStruct;  myStruct->doSomething();  } |

| **Compliant Code** |
| --- |
| Here the two statements are switched so that myStruct is first dereferenced and then the allocated memory is deallocated. |
| struct MyStruct {  void doSomething();  }  void myFunction() {  MyStruct \*myStruct = new MyStruct;  myStruct->doSomething();  delete myStruct;  } |

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

| **Principles(s):**  *2. Heed Compiler Warnings* – This standard maps to this principle because the compiler will warn of this error. The code will still compile.  *3. Architect and Design for Security Policies* – This standard maps to this principle because building the system around secure policies should prevent this error from ending up in the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppDepend | v2023.1 | “MEM50-CPP:Do not access freed memory”  https://www.cppdepend.com/cert-cpp |  |
| PVS Studio | 7.16 | “The ‘Foo’ function is called twice to deallocate the same resource”  https://pvs-studio.com/en/pvs-studio/sast/cert/ |  |
| Clang static Analyzer | 15 | “1.1.2.2. cplusplus.NewDelete (C++)”  https://clang.llvm.org/docs/analyzer/checkers.html#default-checkers |  |
| klocwork | 2020.1 | “UFM.USE.MUST”  https://help.klocwork.com/current/en-us/concepts/candccheckerreference.htm | Good for large codebases and custom checkers, covers security |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Avoid Side Effects in Arguments to Unsafe Macros** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | The assert macro should not receive expressions that have side effects as arguments. If NDEBUG is undefined then assert will evaluate the expression argument and call abort() if the expression equals 0. If NDEBUG is defined then assert will do nothing and the expression is not evaluated. |

| **Noncompliant Code** |
| --- |
| In this example the assertion has a side effect because of the use of the ++ operator. If this is not evaluated then this has implications for the rest of the program. |
| void myFunction(size\_t number) {  assert(number++ > 0);  // function continues...  } |

| **Compliant Code** |
| --- |
| The assertion now has no side effects. The function will run as intended because the side effect has been placed outside of assert. |
| void myFunction(size\_t number) {  assert(number > 0);  ++number;  // function continues...  } |

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

| **Principles(s):**  *2. Heed Compiler Warnings* – This standard maps to this principle because the compiler will warn of this error.  *3. Architect and Design for Security Policies* – This standard maps to this principle because if the developers are building the system with security policies in mind then they should catch this error.  *10. Adopt a Secure Coding Standard* – This standard maps to this principle because a secure coding standard can outline ways that assertions should and should not be used. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | ‘Warn if there are side effects in assert statements”  https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/ | Open-source, cross-platform, free |
| Clang-Tidy | 17.0.0git | “bugprone-assert-side-effect”  https://clang.llvm.org/extra/clang-tidy/checks/list.html | Clang-based C++ linter tool |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle All Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Whenever an exception is thrown, control will pass to the matching handler. If no matching handler is found then std::terminate() is called causing abnormal process termination. |

| **Noncompliant Code** |
| --- |
| In this example myFunction() calls divide() which throws an exception. If 0 is passed to divide() as the denominator, an exception will be thrown but there is no handler, resulting in undefined behavior. |
| float divide(float num, float denom) {  if (denom == 0) {  throw std::invalid\_argument(“Cannot divide by 0”);  }  return num / denom;  }  void myFunction() {  divide(10, 0);  } |

| **Compliant Code** |
| --- |
| Here there is an appropriate handler to catch the exception. |
| int divide(float num, float denom) {  if (denom == 0) {  throw std::invalid\_argument(“Cannot divide by 0”);  }  return num / denom;  }  int myFunction() {  try {  divide(10, 0);  }  catch (const std::invalid\_argument& e) {  std::cout << e.what() << std::endl;  }  } |

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

| **Principles(s):**  *3. Architect and Design for Security Policies* – This standard maps to this principle because implementing exceptions and exception handling will be a part of the system design. This helps to ensure that all exceptions get handled.  *4. Keep it Simple* – This standard maps to this. principle because avoiding complexities in how exceptions are utilize will prevent the code from getting overly complicated, which can make it difficult to see when unhandled exceptions are present.  *8. Practice Defense in Depth* – This standard maps to this principle because exceptions and exception handling can be part of adding security into a system. Properly practicing DID would mean implementing exceptions and handling them appropriately. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | “Unhandled exception specification when calling function foo()”  https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/ | Open-source, cross-platform, free |
| CppDepend | v2023.1 | “ERR-CPP: Handle all Exceptions”  https://www.cppdepend.com/cert-cpp |  |
| Just My Code | Visual Studio 2022 | No name for the rule or check | Visual Studio Debugger |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do Not Read Uninitialized Memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-008-CPP] | If no initialization is performed on an object, it retains an indeterminate value until it is replaced by initialization. If the indeterminate value is read, undefined behavior is produced. |

| **Noncompliant Code** |
| --- |
| Here a variable is declared but not initialized. The program prints the uninitialized variable’s value, leading to undefined behavior. |
| void main() {  char letter;  std::cout << letter << std::endl;  } |

| **Compliant Code** |
| --- |
| Now the variable is initialized, replacing the indeterminate value and avoiding undefined behavior. |
| void main() {  char letter = ‘a’;  std::cout << letter << std::endl;  } |

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

| **Principles(s):**  *2. Heed Compiler Warnings* – This standard maps to this principle because the compiler will warn of uninitialized variables.  *3. Architect and Design for Security Policies* – This standard maps to this principle because while designing the system it should be clear when variables are declared but never used.  *4. Keep it Simple* – This standard maps to this principle because it can be easy to make this mistake in overly complicated code with an unnecessarily high number of variables. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | “Using allocated data before it has been initialized”  https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/ | Open-source, cross-platform, free |
| PVS Studio | 7.16 | “Use of uninitialized variable ‘Foo’”  https://pvs-studio.com/en/pvs-studio/sast/cert/ |  |
| CppDepend | v2023.1 | “EXP-33:Do not read uninitialized memory”  https://www.cppdepend.com/cert-c |  |
| klocwork | 2020.1 | “UNINIT.STACK.MUST”  https://help.klocwork.com/current/en-us/concepts/candccheckerreference.htm | Good for large codebases and custom checkers, covers security |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do Not Attempt to Modify String Literals** |
| --- | --- | --- |
| **String Correctness** | [STD-009-CPP] | Modifying a string literal will cause undefined behavior because they are stored in read-only memory. |

| **Noncompliant Code** |
| --- |
| The pointer myString holds the address of a string literal, and modifying it will cause undefined behavior. |
| void main() {  char \*myString = “hello world”;  myString[0] = ‘H’;  } |

| **Compliant Code** |
| --- |
| Using an array to initialize the string literal creates a copy of the string literal in the space allocated for the array, and therefore is safe to modify. |
| void main() {  char myString[] = “hello world”;  myString[0] = ‘H’;  } |

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

| **Principles(s):**  *3. Architect and Design for Security Policies* – This standard maps to this principle because attempting to modify a string literal should be caught when designing the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppDepend | v2023.1 | “STR30-C: Do not attempt to modify string literals”  https://www.cppdepend.com/cert-c |  |
| klocwork | 2021.1 | “CXX.OVERWRITE\_CONST\_CHAR’  https://help.klocwork.com/current/en-us/concepts/candccheckerreference.htm | Good for large codebases and custom checkers, covers security |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Honor Exception Specifications** |
| --- | --- | --- |
| **Exceptions** | [STD-010-CPP] | If there is exception behavior that does not match the exception-specification provided, this can cause implementation-defined program termination. |

| **Noncompliant Code** |
| --- |
| In this example the function uses the noexcept(true) exception specification, claiming that it will not throw an exception. However, it is possible for std::list::insert() to throw a bad\_alloc exception. |
| void makeList() noexcept(true) {  std::list<int> mylist;  std::list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 5; i++)  mylist.push\_back(i);  it = mylist.begin();  ++it;  mylist.insert(it,10);  } |

| **Compliant Code** |
| --- |
| In this code the noexcept(true) exception specification has been removed, which allows exceptions to be thrown from the function. |
| void makeList() {  std::list<int> mylist;  std::list<int>::iterator it;  // set some initial values:  for (int i = 1; i <= 5; i++)  mylist.push\_back(i);  it = mylist.begin();  ++it;  mylist.insert(it,10);  } |

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

| **Principles(s):**  *2. Heed Compiler Warnings* – This standard maps to this principle because the compiler will warn if a function is assumed to not throw an exception, but does.  *3. Architect and Design for Security Policies* – This standard maps to this principle because architecting and designing for security policies will help to prevent exception mistakes before they happen.  *8. Practice Defense in Depth* – This standard maps to this principle because exceptions and exception handling can be part of adding security into a system. Properly practicing DID would mean implementing exceptions and handling them appropriately. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | “Throwing exception in noexcept, nothrow(), attribute((nothrow)) or \_\_declspec(nothrow) function”  https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/ | Open-source, cross-platform, free |

### 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.

Overall, security should be built into the entire process and should include tasks like choosing safe third-party tools, static and dynamic code analysis, automated tests, and threat modeling. When it comes to implementing automation tools to help enforce the standards in this policy, one area that this can be included is when developers are building the code. Choosing tools that integrate into the development environment will help mitigate threats early on in the process. Another area that automation can be implemented is during the “monitor and detect” section, where automation tools can help collect data and respond to security threats.

### 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 | Low | Unlikely | Medium | Low | 3 |
| STD-002-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | Low | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | High | Probable | Medium | High | 1 |
| STD-009-CPP | Low | Likely | Low | Medium | 2 |
| STD-010-CPP | Low | Likely | Low | Medium | 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 in rest | This type of encryption helps to protect data that is stored on disk, backup media, or a database. This is typically done with an algorithm and is used because it provides another layer of security. There are often other protections in place for data at rest but encryption helps protect even further by translating the data into a form that malicious users cannot decrypt. This applies because a lot of sensitive data is kept and should be practiced at all times since there is always going to be data in rest. This includes protecting files and hard drives. |
| Encryption at flight | This type of encryption refers to data that is in motion over a network connection. Data may be safe at one end or another but during transmission it must be ensured that it remains protected. This is accomplished by using an encryption algorithm on the information before it is sent and is done so that any intercepted data cannot be read. This policy applies because connections are used to send sensitive data and it is important that this data is kept safe, even if it is compromised. This should be practiced whenever information is sent over any connection. This includes protecting emails and websites. |
| Encryption in use | This type of encryption refers to protecting data that is currently being accessed or read. Data may be protected while in rest or while being transported, but data in use is may be stored as plain text for the time that it is being used. This can be applied using encryption methods that allow data to be used while still keeping it encrypted. This policy applies because using unprotected data is a weak point that attackers will want to exploit, especially if other areas are well protected and in-use data is the most vulnerable area. This should be used whenever data is being read or accessed, especially if that data is usually encrypted when at rest or in motion. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying that a user is who they say they are by using methods such as requiring usernames, passwords, biometrics, or multi-factor authentication when a user logs into a system. This policy applies because many different users will need to use a system and there are several different user types that will need to access various resources. Different levels of user access means that different permissions will be granted depending on who the user is. It is important to only allow users to access the resources that they are authorized to use, and the first step in accomplishing this is to authenticate every user before they are granted access to the system. |
| Authorization | Authorization refers to making sure that resources or areas of a network are only accessible by users with the proper authentication. For example, there may be files that all users are allowed to access, but certain more sensitive files should only be accessed by users of a higher level, such as admins. This is also the process of enforcing security policies. Beyond ensuring that users are authenticated, authorization can be accomplished with techniques such as restrictions (location, time of day, login frequency, multiple logins), filtering IP addresses, managing bandwidth traffic, and using encryption methods. This policy applies because of the many different users that will need permissions for various resources, and should be practiced whenever system resources are being accessed. This also includes the addition of new users who will need to be authorized for different tasks and data. |
| Accounting | Accounting is the process of keeping records of information about the system and how users interact with it, including measuring resource consumption, logging session information, and monitoring information that is being sent or received. This policy applies because it helps to keep track of what was done, who did it, and when. For example, making changes to a database can lead to serious consequences and the company may need to know who made the change and why. This should be practiced whenever anything in the system changes so that accurate historical records are kept. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 04/09/2023 | Updated Policy | Jordan Ballard | Brian Enochson |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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