**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 | Input data validation is important to security because improperly validated inputs can be used by attackers to compromise systems by means of SQL injection, buffer overflows, and cross-site scripting. To prevent these vulnerabilities data must follow proper guidelines for format, lengths, and ranges. |
| 1. Heed Compiler Warnings | Compiler warnings can point out type mismatches, uninitialized variables, or deprecated functions to bring awareness to potential sources of vulnerabilities. To improve the code’s safety and maintainability these warnings should always be addressed. |
| 1. Architect and Design for Security Policies | Security policies like access controls, authentication mechanisms, and encryption should always be considered when designing software. A proactive approach in the design process makes it significantly easier to enforce security throughout the software lifecycle rather than adding in security measures later in the process. |
| 1. Keep It Simple | The more difficult a system is to understand, the more difficult it is to keep it secure. When the design of the code is simple you can reduce the rate of security vulnerabilities being overlooked due to misunderstandings or incorrect configurations. |
| 1. Default Deny | With a default deny security posture, the system will not allow any actions that are not explicitly permitted by the system. This is done to try and minimize any instanced of unauthorized access or action and to provide a level of safety that is higher than a default allow stance. |
| 1. Adhere to the Principle of Least Privilege | To limit the damage from accidental mishaps as well as any malicious activity, the system, process, or user should be given the least set of privileges necessary to accomplish their tasks. |
| 1. Sanitize Data Sent to Other Systems | Any time you have data leaving your system to interact with other systems you should take caution to sanitize the data to remove or encode any potentially harmful elements. This is done to prevent any kind of attacks involving data injection or data leaks. |
| 1. Practice Defense in Depth | Defense in Depth should always be practiced ensuring multiple layers of security are present in the IT environment. This practice will always make it more difficult for an attacker to breach your system because they need to crack multiple layers of security. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance is for more than just functionality and performance because it also helps to ensure security as well. When you implement techniques including code reviews, penetration testing, and automated security scanning, you can identify and resolve any problems present before software goes into deployment. |
| 1. Adopt a Secure Coding Standard | Secure coding standards give developers guidelines to work within to help write secure code. Standards like SEI CERT should be used to provide a set of rules that can be followed to reduce many common vulnerabilities in code. |

### 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** | **Pass an object of the correct type to va\_start** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Passing an object of array type still produces undefined behavior in C++ because an array type as a function parameter requires the use of a reference, which is prohibited. Additionally, passing an object of a type that undergoes default argument promotions still produces undefined behavior in C++. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the object passed to va\_start() will undergo a default argument promotion, which results in undefined behavior. |
| #include <cstdarg>    **extern** "C" **void** f(**float** a, ...) {  **va\_list** list;  **va\_start**(list, a);    // ...  **va\_end**(list);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f() accepts a double instead of a float. |
| #include <cstdarg>    **extern** "C" **void** f(**double** a, ...) {  **va\_list** list;  **va\_start**(list, a);    // ...  **va\_end**(list);  } |

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

| **Principles(s):**  1.Validate Input Data: Ensuring correct types when passing objects to functions is an essential form of input validation.  10.Adopt a Secure Coding Standard: By ensuring correct object types are passed, we adhere to secure coding practices, reducing potential vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 13.0.0 | Unix.VA-arg | Detects misuse of variable argument lists |
| GCC | 11.2 | -Wvarargs | Detects variable arguments issues |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do no cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| **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):**  1.Validate Input Data: Validating that a value is within the enumeration's range before casting is crucial for system security.  4.Keep It Simple: Avoiding unnecessary type casts maintains a simpler and more understandable system design.  8.Practice Defense in Depth: This coding standard acts as one of the layers in a defense-in-depth strategy.  10. Adopt a Secure Coding Standard: Ensuring valid enumeration ranges aligns with best practices for secure coding. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 13.0.0 | core.CastToStruct | Checks for casting to structure type |
| GCC | 13.2.0 | -Wenum-conversion | Emits a warning when a value is cast to an enumeration type that cannot represent it |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not call any of the preceding functions with a null pointer as the const charT \* argument. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #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):**  1.Validate Input Data: Ensuring that pointers are not null before using them is an essential form of input validation, preventing potential software crashes.  10.Practice Defense in Depth: By validating pointers, this coding standard acts as an additional layer of defense, making it harder for vulnerabilities to be exploited. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 13.0.0 | core.NonNullParamChecker | Checks for potential null pointer dereferences |
| GCC | 13.2.0 | -Wnull-dereference | Gives a warning when a null pointer is dereferenced |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted. |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system(). |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **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):**  1.Validate Input Data: Sanitizing data before it's passed to complex subsystems is a critical form of input validation, ensuring that potentially harmful data doesn't get executed.  7.Sanitize Data Sent to Other Systems: This principle directly underlines the importance of ensuring data passed to another system or subsystem is cleaned of any malicious elements.  8.Practice Defense in Depth: By sanitizing input data, this coding standard provides an additional layer of defense against potential attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.10.0 | Active Scan | Detects possible SQL injection vulnerabilities |
| SQLMap | 1.5.0 | Automated SQL Injection detection | Detects and exploits SQL Injection flaws |
| Fortify Static Code Analyzer | 21.1.0 | SQL Injection | Identifies potential SQL points |
| GCC | 13.2.0 | -Wformat-security | Warns when a string is passed to a function that expects a formatted string but string is not sanitized |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to an 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** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #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):**  1.Validate Input Data: Checking the result of memory allocations is a form of output validation, ensuring that the software isn't working with invalid memory references.  4.Keep It Simple: Straightforward error checking and handling make the software more maintainable and predictable.  8.Practice Defense in Depth: By validating memory allocations, we add an additional layer of defense against potential system crashes or vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.6 | General Analysis | Check for various issues |
| Valgrind | 3.1.70 | Memcheck | Detects memory errors |
| Clang Static Analyzer | 13.0.0 | alpha.cplusplus.NewDelete | Detects mismatches between memory allocation and deallocation |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Do not abruptly terminate the program** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Do not explicitly or implicitly call std::quick\_exit(),  std::abort(), or std::\_Exit(). When the default terminate\_handler is installed or the current terminate\_handler responds by calling std::abort() or std::\_Exit(), do not explicitly or implicitly call std::terminate(). Abnormal process termination is the typical vector for denial-of-service attacks. |

| **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):**  1.Validate Input Data: Ensuring proper validation and handling at every stage can mitigate abrupt program termination.  4.Keep It Simple: Clear exception handling and program flow documentation helps in maintaining and understanding the software's behavior.  8.Practice Defense in Depth: Preventing abrupt termination is an essential layer in a multi-tiered defense strategy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.6 | General Analysis | Checks for issues |
| Clang Static Analyzer | 13.0.0 | unix.Malloc, unix.MismatchedDeallocator, c++ NewDelete | Checks for issues that may lead to sudden termination |
| Coverity | 2023 | RESOURCE\_LEAK, EXCEPT | Checks for potential sudden terminations |
| GCC | 13.2.0 | -Wuncaught-exception | Warning when exception not caught |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **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):**  1.Validate Input Data: Catching and handling exceptions ensures unexpected or erroneous data or states do not lead to unintended program behavior.  4.Keep It Simple: Structured exception handling simplifies the debugging process and ensures errors are handled uniformly.  8.Practice Defense in Depth: Catching all exceptions ensures that even if one layer of the application encounters an issue, it doesn't lead to the application crashing or behaving unpredictably. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.6 | General Analysis | Check for issues |
| GCC | 13.2.0 | -Wuncaught-exception | Warning when exception not caught |
| Clang Static Analyzer | 13.0.0 | core.Exceptions | Checks for issues with exception handling |
| Coverity | 2023 | EXCEPTION\_HANDLING | Checks for issues with exception handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and null terminator** |
| --- | --- | --- |
| **Memory Protection** | [STD-008-CPP] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ std::basic\_string template requires no such character. |

| **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;  } |

| **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):**  1.Validate Input Data: Ensuring safe storage space effectively validates the size of input data, preventing buffer overflows.  4.Keep It Simple: Using higher-level abstractions like std::string reduces the risk of vulnerabilities by abstracting away complexity.  8.Practice Defense in Depth: By ensuring storage safety, you're adding a layer of security against many common attack vectors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.6 | bufferOverrun | Detects buffer overrun |
| Clang Static Analyzer | 13.0.0 | core.BufferOverflow | Checks for buffer overflows |
| GCC | 13.2.0 | -Wstringop-overflow | Warning when string operation could lead to buffer overflow |
| AddressSanitizer |  | General Memory Safety | Detects memory issues |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Range check element access** |
| --- | --- | --- |
| **String Correctness** | [STD-009-CPP] | The std::string index operators const\_reference operator[](size\_type) const and reference operator[](size\_type) return the character stored at the specified position, pos. When pos >= size(), a reference to an object of type charT with value charT() is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the value returned by the call to get\_index() may be greater than the number of elements stored in the string, resulting in undefined behavior. |
| #include <string>    **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");    s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the std::basic\_string::at() function, which behaves in a similar fashion to the index operator[] but throws a std::out\_of\_range exception if pos >= size(). |
| #include <stdexcept>  #include <string>  **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");  **try** {      s.at(get\_index()) = '1';    } **catch** (std::out\_of\_range &) {      // Handle error    }  } |

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

| **Principles(s):**  1.Validate Input Data: By checking range on string element access, we validate the input indices preventing potential out-of-bounds accesses.  4.Keep It Simple: Using std::basic\_string::at(), a simple and safer way to access string elements, reduces complexity and potential errors.  8.Practice Defense in Depth: Ensuring every string access is within bounds acts as a layer of defense against potential vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.6 | arrayIndexOutOfBounds | Detects out-of-bounds array accesses |
| Clang Static Analyzer | 13.0.0 | core.ArrayBound | Checks for out-of-bounds array accesses |
| AddressSanitizer |  | Bounds-checking | Detect out-of-bounds accesses in real-time |
| GCC | 13.2.0 | -Wstringop-overread | Warning when string operation could read out of bounds |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Never qualify a reference type with a const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-010-CPP] | Do not attempt to cv-qualify a reference type because it results in undefined behavior. A conforming compiler is required to issue a diagnostic message. However, if the compiler does not emit a fatal diagnostic, the program may produce surprising results, such as allowing the character referenced by p to be mutated. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a const-qualified reference to a char is formed instead of a reference to a const-qualified char. |
| #include <iostream>    **void** f(**char** c) {  **char** &**const** p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the const qualifier. |
| #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):**  1.Validate Input Data: By checking range on string element access, we validate the input indices preventing potential out-of-bounds accesses.  4.Keep It Simple: Using std::basic\_string::at(), a simple and safer way to access string elements, reduces complexity and potential errors.  8.Practice Defense in Depth: Ensuring every string access is within bounds acts as a layer of defense against potential vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.6 | arrayIndexOutOfBounds | Detects out-of-bounds array accesses |
| Clang Static Analyzer | 13.0.0 | core.ArrayBound | Check for out-of-bounds array accesses |
| AddressSanitizer |  | Bounds-Checking | Detects out-of-bounds accesses during runtime |
| GCC | 13.2.0 | -Wqual-type-qualifier | Warning when a reference type is qualified with const or volatile |

### 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.

In the DevSecOps model, the integration of security starts at the source control stage, where pre-commit hooks verify code compliance to industry standards. During the Continuous Integration phase, static code analysis tools are used to check the codebase, and any violations will halt the build process. Before deployment in the Continuous Deployment stage, dynamic analysis validates application adherence to standards. The Infrastructure as Code provisions environments that bolster these checks. After deployment, continuous monitoring tools will detect runtime violations, providing instant feedback. If any post-deployment violations occur, a clear incident response plan will aid in swift remediation, while continuous security training will help the development team remain up to date on the best practices.

### 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 | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Low | High | 4 |
| STD-003-CPP | High | Likely | Low | High | 4 |
| STD-004-CPP | High | Likely | Medium | High | 4 |
| STD-005-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-006-CPP | Medium | Unlikely | High | Medium | 2 |
| STD-007-CPP | High | Unlikely | High | High | 3 |
| STD-008-CPP | High | Likely | Low | High | 4 |
| STD-009-CPP | High | Likely | Medium | High | 4 |
| STD-010-CPP | Medium | Unlikely | Medium | 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 | Encryption of data that is stored in any kind of storage medium including a database, disk, tape, etc. By encrypting these mediums, you can ensure that if the device is stolen or accessed without authorization, the data will stay safe. |
| Encryption at flight | Encryption of data being transferred over networks. This is done using secure protocols like HTTPS, SSL/TLS, or VPNs. This is done to make sure data confidentiality and integrity remains safe during transfers. |
| Encryption in use | Encryption of data currently being used. This is done by ensuring data remains encrypted while in memory or during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Process where users prove their identity. Generally done through usernames and passwords but can use other means like biometrics or multi-factor authentication. This helps to make sure only legitimate users can gain access. |
| Authorization | This process decides what levels of access authenticated users have and what they can do. This is accomplished by setting roles and permissions that set actions available to the users. This is done to make sure users can only have access to what they are meant to have access to. |
| Accounting | Process that tracks user activities and makes a record of those activities. This is done by logging the actions of a user as soon as they log on, what actions they took and what resources they accessed. This is done to keep a trail of activities that can be audited so that changes to the database can be tracked, malicious activities can be detected, or to help diagnose issues. |

**\***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 | 09/19/2023 | Initial Template | Michael Antoniazzi |  |
| 2.0 | 10/14/2023 | Completed Template | Michael Antoniazzi | [Insert text.] |
| [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 |