**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 | Do not trust any external sources when it comes to data validation. Proper validation reduces tremendous amounts of vulnerabilities. |
| 1. Heed Compiler Warnings | When creating and compiling code, use the highest warning settings available and modify code to eliminate these warnings. These tools are incredibly useful in detecting security risks that the developer may not have otherwise noticed. Heeding the warnings and modifying the code creates a more secure program. |
| 1. Architect and Design for Security Policies | Design the structures of your code with security policies in mind. By implementing an architecture that enhances security in the initial design and development phases, you save the company a lot of time and money in correcting issues found in later testing and deployment phases. |
| 1. Keep It Simple | Complex code systems have more room for errors, and therefore more vulnerabilities. By keeping the code simple, you maximize the amount of time and energy spent on correcting, securing, and maintaining an efficient system. |
| 1. Default Deny | The default setting should be to deny access to the system. Set conditions that have to be met before permission is granted. |
| 1. Adhere to the Principle of Least Privilege | Ensure that users only have access to code functions and data that would be needed to complete their task. By giving them the minimum amount of access needed, you protect the rest of the system from corruption. |
| 1. Sanitize Data Sent to Other Systems | When data is sent to other systems, it crosses a “trust boundary”, and needs to be protected to prevent data leaks or exposure. This data can easily be compromised as the system it is being sent to doesn’t necessarily understand the context of the data it is receiving. Implementing sanitizing techniques protects transmitted data. |
| 1. Practice Defense in Depth | Implement multiple layers of defensive systems so that even in one is corrupted or proves to be inaccurate, there are several other layers of defense in place to protect the system from vulnerability. |
| 1. Use Effective Quality Assurance Techniques | Implementing effective quality assurance techniques in the early development phases helps assess for weak or vulnerable sections of the system. Multiple testing techniques should be incorporated to ensure the system is sound. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard means to incorporate the security principles into your design and development process and create a system that is not vulnerable to attacks or corruption. Security should be a main focus throughout all phases of development. |

### 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. The arithmetic value being case must be within the range of values the enumeration can represent to avoid operating on unspecified values. Checking for out-of-range values must be performed before cast expression. |

| **Noncompliant Code** |
| --- |
| This code attempts to check whether a given value is within range of acceptable values, however, it does so after casting to the enumeration type. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| Compliant code checks that the value can be represented by the enumeration type before the conversion. This guarantess that the conversion does not result in an unspecified 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):** Validate Input Data – Casting to out or range values does not allow for proper input data validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Cast-integer-to-enum | Partially Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 |  |
| CodeSonar | 8.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion alters value  Cast alters value |
| HelixQAC | 2023.3 | C++3013 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Value-returning functions must return a value from all exit paths. Otherwise, it will result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The programmer does not return the input value for positive input, so any number greater than or equal to 0 does not return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |

| **Compliant Code** |
| --- |
| By adding a return outside of the if statement, all inputs will return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |

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

| **Principles(s):** Validate Input Data = Validating data ensures accuracy |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Return-impicit | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC52 |  |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as function-try-blocks |
| CodeSonar | 8.0p0 | LANG.STRUCT.MRS | Missing return statement |

#### 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. Passing a null pointer to this function is undefined behavior as it would result in dereferencing a null pointer. |

| **Noncompliant Code** |
| --- |
| Std::string is created from the result of calling std::getenv(). But std::getenv() returns a null pointer on failure, so this could lead to undefined behavior when the environment variable does not exist. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| As a 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):** Architect and Design for Security Policies - Derefencing a null pointer could be exploited and result in execution of arbitrary code, a serious vulnerability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C++:STR51-CPP | Checks for string operations on null pointer (rule partially covered) |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Astree | 22.10 | Assert\_failure |  |
| Klocwork | 2023.3 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |

#### Coding Standard 4

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

| **Noncompliant Code** |
| --- |
| Two unrelated smart pointers are constructed from the same underlying pointer value. When the local variable p2 is destroyed, it deletes the pointer value in manages. Then, when p1 is destroyed, it deletes the same pointer value resulting in a double-free vulnerability. |
| #include <memory>    **void** f() {  **int** \*i = **new** **int**;    std::shared\_ptr<**int**> p1(i);    std::shared\_ptr<**int**> p2(i);  } |

| **Compliant Code** |
| --- |
| Std::shared\_ptr objects are related to one another through copy construction. When p2 is destroyed, the use count for the pointer value is decremented but still nonzero. This compliant solution also calls std::make\_shared() instead of allocation a raw pointer and storing its value in a local variable. |
| #include <memory>    **void** f() {    std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();    std::shared\_ptr<**int**> p2(p1);  } |

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

| **Principles(s):** Architect and Design for Security Policies – Upholding this standard prevents double-free vulnerability and undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.29 | V1006 |  |
| Polyspace Bug Finder | R2023b | CERT C++:MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |
| Astree | 22.10 | Dangling\_pointer\_use |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM56-a | Do not store an already owned pointer value in an unrelated smart pointer |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Detect and handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| Two memory allocations are performed within the same expression. Because memory allocations are passed as arguments to a function call, an exception thrown as a result of one of the calls to new could result in a memory leak. |
| **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(A \*, B \*);  **void** f() {    g(**new** A, **new** B);  } |

| **Compliant Code** |
| --- |
| Std::unique\_ptr is used to manage resources for A and B objects, preventing the errors described in the noncompliant code. |
| #include <memory>    **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(std::unique\_ptr<A> a, std::unique\_ptr<B> b);  **void** f() {    g(std::make\_unique<A>(), std::make\_unique<B>());  } |

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

| **Principles(s):** Architect and Design in Security Principles – Failing to detect and handle memory allocation issues can result in denial of service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.29 | V522, V668 |  |
| Polyspace Bug Finder | R2023b | CERT C++:MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| Parasoft Insure++ |  |  | Runtime detection |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because of the order of evaluation of a function’s parameters is undefined |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression. Assertions are valuable tools in finding and eliminating defects that may result in vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. It needs to be placed in a function and 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** |
| --- |
| Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. |
| #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):** Use Effective Quality Assurance Techniques – Using static assertions identifies vulnerabilities instead of leaving silent malfunctions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |
| CodeSonar | 8.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| 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. All exceptions thrown must be caught by a matching exception handler. This ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **Noncompliant Code** |
| --- |
| neither f() or 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** |
| --- |
| The main entry point handles all exceptions, which ensures the stack is unwound up to the main() function. |
| **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):** Adopt a Secure Coding Standard – Exception handling is essential to creating a functional system and should be upheld as a secure coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| CodeSonar | 8.0p0 | LANG.STRUCT.UCTCH | Unreachable catch |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Polyspace Bug finder | R2023b | CERT C++:ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory | [STD-008-CPP] | Do not access freed memory. When memory is freed, all pointers into it become invalid. The data at the freed location can appear valid but change unexpectedly. |

| **Noncompliant Code** |
| --- |
| S is dereferenced after it has been deallocated. This vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

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

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

| **Principles(s):** Practice Defense in Depth – Accessing a dangling pointer can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ |  |  | Runtime detection |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exceptions | [STD-009-CPP] | Guarantee exception safety. Proper handling of errors and exceptional situations is essential for continued correct operation of a system. |

| **Noncompliant Code** |
| --- |
| No exception safety. This code shows a flawed copy assignment operator, which will ultimately result in undefined behavior if the new expression throws an exception. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }        IntArray(**const** IntArray& that); // nontrivial copy constructor    IntArray& operator=(**const** IntArray &rhs) {  **if** (**this** != &rhs) {  **delete**[] array;        array = nullptr;        nElems = rhs.nElems;  **if** (nElems) {          array = **new** **int**[nElems];          std::**memcpy**(array, rhs.array, nElems \* **sizeof**(\*array));        }      }  **return** \***this**;    }      // ...  }; |

| **Compliant Code** |
| --- |
| This copy assignment operator provides the strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }      IntArray(**const** IntArray& that); // nontrivial copy constructor      IntArray& operator=(**const** IntArray &rhs) {  **int** \*tmp = nullptr;  **if** (rhs.nElems) {        tmp = **new** **int**[rhs.nElems];        std::**memcpy**(tmp, rhs.array, rhs.nElems \* **sizeof**(\*array));      }  **delete**[] array;      array = tmp;      nElems = rhs.nElems;  **return** \***this**;    }      // ...  }; |

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

| **Principles(s):** Adopt a Secure Coding Standard – Proper exception handling is essential in maintaining a secure, functioning system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | ALLOC.LEAK | Leak |
| LDRA toold suite | 9.7.1 | 527 S, 56 D, 71 D | Partially Implemented |
| Polyspace Bug Finder | R2023b | CERT C++: ERR%^-CPP | Checks for exceptions violating class invariant (rule fully covered) |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions  Do not leave catch blocks empty |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory | [STD-010-CPP] | Guarantee that library functions do not overflow. Copying data into a container that is not large enough to hold it all results in a buffer overflow, resulting in vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The std::copy() algorithm has no inherent bounds checking, resulting in a buffer overflow on copying the first element. |
| #include <algorithm>  #include <vector>    **void** f(**const** std::vector<**int**> &src) {    std::vector<**int**> dest;    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

| **Compliant Code** |
| --- |
| This iterator expands the container by one element for each element supplied by the algorithm, guaranteeing the container will be large enough to hold all of the elements provided. |
| #include <algorithm>  #include <iterator>  #include <vector>    **void** f(**const** std::vector<**int**> &src) {    std::vector<**int**> dest;    std::copy(src.begin(), src.end(), std::back\_inserter(dest));    // ...  } |

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

| **Principles(s):** Keep It Simple – By creating code that ensures that the container is large enough to hold all of the elements, buffer overflow vulnerabilities are prevented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | BADFUNC.BO.\*  LANG.MEM.BO  LANG.MEM.TBA | A collection of warning classes that report uses of library functions prone to internal buffer overflows.  Buffer Overrun  Tainted Buffer Access |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-CTR52-a | Do not pass empty container iterators to std algorithmsas destinations |
| Polyspace Bug Finder | R2023b | CERT C++:CTR52-CPP | Checks for library functions overflowing sequence container (rule partially covered) |
| Astree | 22.10 | Invalid\_pointer\_dereference |  |

### 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.

While DevOps already has a well-implemented process in place, adding automation to the verify and test phase will significantly improve the existing process. Implementing automated testing will allow developers to test their code more efficiently with minimal interaction. These automated tests will protect a project from a variety of security vulnerabilities and implementation of inefficient code. Automation also ensures that each developer’s code is held to the same standard and testing.

### 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 | 4 | 3 |
| STD-002-CPP | Medium | Probable | Medium | 8 | 2 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CPP | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Unlikely | High | 1 | 3 |
| STD-007-CPP | Low | Probable | Medium | 4 | 3 |
| STD-008-CPP | High | Likely | Medium | 18 | 1 |
| STD-009-CPP | High | Likely | High | 9 | 2 |
| STD-010-CPP | High | Likely | Medium | 18 | 1 |

### 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 in rest is the process of translating saved data to another form of data via encryption. It allows for the protection of sensitive data and requires authorization to decrypt the information. |
| Encryption at flight | Encryption at flight is the process of encrypting data that is moving over a network. This policy is especially important when moving data over the internet. It protects sensitive data from being breached during a data transfer process. |
| Encryption in use | Encryption in use is the process where data is never left unsecured regardless of it lifecycle stage. Data access is monitored and controlled through authorization during this approach. Data requests are analyzed and responded to in real tie allowing awareness of any suspicious activity. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The authentication process is where users are confirmed through proper credentials (usernames, passwords, two-factor authentication and biometrics) to gain access to a system. This policy is especially effective in controlling unauthorized access to user accounts. |
| Authorization | The authorization process is where levels of access are determined for system users. Depending on what type of user is accessing the system, limitations to access can be established appropriately. This approach would enable only those with administrator privileges to gain access to databases and employee records, make changes to those records, and add new users to the system. |
| Accounting | The accounting process is where system users are monitored and logged according to user interaction. This approach allows administrators to keep record of when data had been accessed or changed in the system and who made the changes. These processes work together to give a system a multi-layer defense. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 02/18/2024 | Initial Report | Laura McAroy |  |
| 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 |