**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 | Data validation is an essential for a variety of reasons. Validating input data allows for ensuring accuracy, mitigation of corrupted data, and preventing harm to a system. |
| 1. Heed Compiler Warnings | Considering compiler warnings is essential in addressing problems and unintended behavior in your code. It’s important to utilize these static and dynamic analysis tools to detect and eliminate security vulnerabilities. |
| 1. Architect and Design for Security Policies | Implementing proper security policies is essential in protecting your system and upholding integrity. This process addresses safeguards and security controls to protect sensitive data. |
| 1. Keep It Simple | This technique promotes the implementation of code to be simple and precise. Adopting the “Keep It Simple” prevents the implementation of an overly complex design where it can be more challenging to address security vulnerabilities. |
| 1. Default Deny | All inbound and outbound data traffic should be blocked unless it has been expressly permitted. |
| 1. Adhere to the Principle of Least Privilege | Access rights should be limited to what a user is needed to perform their job duties. This approach limits access of data to only what is necessary for the specified user. |
| 1. Sanitize Data Sent to Other Systems | All data input should be monitored, filtered, and cleaned. This process provides protection from a variety of cyberattacks and upholds integrity of the system. |
| 1. Practice Defense in Depth | Utilizing a Defense in Depth approach allows for a well layered security. This layered defense approach assists in guarding from and containing security threats. |
| 1. Use Effective Quality Assurance Techniques | Utilizing quality assurance techniques can be effective in the developmental phase. Performing a variety of testing techniques on your system allows for the removal of defective and vulnerable components. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves utilizing all of the core security principles. Applying these standards assists in creating a protected and a non-defected system. |

### 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-CCP | Do not cast to an out-of-range enumeration value: Avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates the attempt to check whether a given value is in acceptable range. The issues is that this check is being performed after casting to the numeration 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** |
| --- |
| This compliant code example demonstrates checking whether a value can be represented by the enumeration type before performing the conversion. |
| **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 unspecific 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** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 |  |
| CodeSonar | 7.1p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2022.3 | C++3013 |  |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CCP | Use valid references, pointers, and iterators to reference elements of a container: References and pointed can be invalidated independently for the same operation. This can result in an invalidated reference but not an invalidated pointer. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates pos being invalidated after the first call to insert(), and subsequent loop iterations have undefined behavior. |
| **void** f(**const** **double** \*items, std::**size\_t** count) {    std::deque<**double**> d;    auto pos = d.begin();  **for** (std::**size\_t** i = 0; i < count; ++i, ++pos) {      d.insert(pos, items[i] + 41.0);    }  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates pos being assigned a valid iterator on each insertion that prevents undefined behavior. |
| **void** f(**const** **double** \*items, std::**size\_t** count) {    std::deque<**double**> d;    auto pos = d.begin();  **for** (std::**size\_t** i = 0; i < count; ++i, ++pos) {      pos = d.insert(pos, items[i] + 41.0);    }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Utilizing proper and valid references prevents undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2022.3 | C++4746,C++4747,C++4748,C++4749 |  |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| Polyspace Bug Finder | R2022b | CERT C++:CTR51-CPP | Checks for use of invalid iterator (rule partially covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for string has sufficient space for character space for character data and the null terminator: Ensuring sufficient size to hold data prevents buffer overflow |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates the input as unbounded that could potentially lead to buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates ensuring data is not truncated and protection against buffer overflow by utilizing std::string instead of a bounded array. |
| #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):** Architect and Design for Security Policies – This standard promotes this policy by ensuring sufficient data storage preventing architect and design issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Helix QAC | 2022.3 | C+2835,C++2836,C++2839,C++5216 |  |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the ‘char’ buffer to store input from  ‘std::cin’ |
| SonarQube C/C++ Plugin | 4.10 | S3519 |  |

#### 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: Prevent double-free vulnerability and undefined behavior. |

| **Noncompliant Code** |
| --- |
| This noncompliant coding example demonstrates two unrelated smart pointers being constructed from the same underlying pointer value. When the local automatic variable 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** |
| --- |
| This compliant coding example demonstrates the std::shared\_ptr objects as related to one another through copy construction. When the automatic variables are destroyed the use count for the shared pointer value is decremented but still nonzero. |
| **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):** Sanitize Data Sent to Other Systems, Architect and Design for Security Policies – Upholding this standard by not storing already-owned pointer values in unrelated smart pointers 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** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Dangling\_pointer\_user |  |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| Polyspace Bug Finder | R2022b | Cert C++: MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |
| Axivion Bauhaus Suite | 7.2.0 | CERTC++-MEM56 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Properly deallocate dynamically allocated resources: Deallocating a pointer that is not allocated dynamically is undefined behavior because the pointer value was not obtained by an allocation function. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates the resulting pointer of the passed expression causing undefined behavior due to the attempt of freeing memory that was not returned. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...    **delete** s1;  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates a solution by instead explicility calling s1’s destructor. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...      s1->~S();  } |

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

| **Principles(s):** Practice Defense in Depth: Not complying to this standard can cause undefined behavior and vulnerabilities potentially breaching established security layers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Invalid\_dynamic\_memory\_allocation  Danagling\_pointer\_use |  |
| Polyspace Bug Finder | R2022b | Cert C++MEM51-CPP | Checks for:  Invalid deletion of pointer  Invalid free of pointer  Deallocation of previously deallocated pointer |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP\_MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets ([]) for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resources |
| SonarQube C/C++ plugin | 4.10 | S1232 |  |

#### 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: Allows for the check of a constant-expressions value. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates of the assert() macro that occurs only at runtime and onlif if the code path containing the assertion is executed. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates the use of a conditional statement to interact with a constant expression allowing for clear diagnostic messages. |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard – Testing expression components promotes quality assurance and upholds a secure coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | misc-static-assert | Check by clang-tidy |
| Éclair | 1.2 | CC2.DCL03 | Fully Implemented |
| Code Sonar | 7.1p0 | (customization) | Users can implement a custom check that reports use of the assert () macro |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Guarantee exception safety: Proper handling of errors and exception situations is essential for the continued correct operation of software. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates a flawed copy assignment operator. IF the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class that can result in undefined behavior. |
| #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 compliant code example demonstrates the copy assignment operator providing the strong exception safety guarantee. The function allocated new storage for the copy before changing the state of the object. |
| **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- Ensuring exception handling is crucial for system operations and should be upheld as a secure coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | AllOC.LEAK | Leak |
| LDRA tool suite | 9.7.1 | 527 S, 56 D, 71 D | Partially implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions  Do not leave ‘catch’ blocks empty |
| Polyspace Bug Finder | R2022b | CERT C++: ERR56-CPP | Checks for exceptions violated class invariant (rule fully covered). |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Management** | STD-008-CPP | Avoid using default operator new for over-aligned types: Utilizing a default operator may result an object being constructed at a misaligned location. This may result in undefined behavior and abnormal termination. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates the new expression being utilized to invoke the default operator new to obtain storage in which to then construct an object of the user-defined type Vector with alignment that exceeds the fundamental alignment of most implementations. These over-aligned objects are typically required by SIMD vectorization instructions that can trap when passed unsuitably aligned arguments. |
| **struct** alignas(32) Vector {  **char** elems[32];  };    Vector \*f() {    Vector \*pv = **new** Vector;  **return** pv;  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates an overloaded operator new function is defined to obtain appropriately aligned storage. |
| #include <cstdlib>  #include <new>    **struct** alignas(32) Vector {  **char** elems[32];  **static** **void** \*operator **new**(**size\_t** nbytes) {  **if** (**void** \*p = std::aligned\_alloc(alignof(Vector), nbytes)) {  **return** p;      }  **throw** std::bad\_alloc();    }  **static** **void** operator **delete**(**void** \*p) {  **free**(p);    }  };    Vector \*f() {    Vector \*pv = **new** Vector;  **return** pv;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Upholding this standard will prevent potential complications that could cause undefined behavior or abnormal termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Default-new-overaligned-type | Fully checked |
| Helix QAC | 2022.3 | C++3120 |  |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM57-a | Avoid using the default operator ‘new’ for over-aligned types |
| Polyspace Bug Finder | R2022b | CERT C++: MEM57-CPP | Checks for situations where operator new is not overloaded for possibly overaligned types (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-009-CPP | Handle all exceptions: all exceptions thrown by an application must be caught by a matching exception handler. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates a situation where no matching handler can be found for the exception thrown. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates the main entry point handling all exceptions. |
| **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 – Catching all exceptions is crucial for system operations and therefore be upheld as a coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Potentially-throwing-static-initialization | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR58 |  |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-010-CPP | Range check element access |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates the possibility of the return value being greater than the number of elmeted stored in the string. This could potentially cause undefined behavior. |
| #include <string>    **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");    s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant code example demonstrates a solution by utilizing the std::basic\_string::at() function. |
| #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):** Keep It Simple – Applying this simple solution allows for protection of potential data storage issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Helix QAC | 2022.3 | C++3162, C++3163, C++3164, C++3165 |  |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
| Astrée | 22.10 | Assert\_failure |  |

### 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.

Utilizing automation through the verify and test phase will be significantly beneficial towards this well-established DevOps 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. Lastly, These automated tests allow for an established approach that can hold an entire projects work to a specific standard.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | High | Probable | High | P6 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | High | Likely | High | P9 | L2 |
| STD-008-CPP | Medium | Unlikely | Low | P6 | L2 |
| STD-009-CPP | Low | Likely | Low | P9 | L2 |
| STD-010-CPP | High | Unlikely | Medium | P6 | L2 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is the process of translating saved data to another form of data through encryption. This process allows for the protection of sensitive data and requires authorization to decode the encrypted information. This policy adds another layer of security to assist with other layers such as firewalls and anti-virus software. |
| Encryption at flight | Encryption at flight is the process of encrypting data that is moving over a network. This policy is especially significant and beneficial in moving data when utilizing open 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 its lifecycle stage (rest, at flight, or in use). Data access is monitored and controlled through authorization during this approach. Data requests are analyzed and responded to in real time allowing awareness 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, biometrics, etc.) to access a system. This policy is especially effective in controlling unauthorized access to user accounts. Applying this approach as a multi-level policy, reinforces and supports system security layers. |
| 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 allows for sensitive data to only be examined by individuals who absolutely need to. |
| Accounting | The accounting process is where system users are monitored and logged according to user interaction. This approach allows administrators to keep track of system components being accessed and how they are being interacted with. |

**\***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 |  |
| [Insert text.] | 11/10/2022 | Revision | Joseph Dugan | [Insert text.] |
| [Insert text.] | 12/2/2022 | Final Revision | Joseph Dugan | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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