**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 from Unverified Sources | Treat all data from external and unverified sources as untrusted until the information has been validated and sanitized. |
| 1. Limit the scope of functions and variables | The smaller the scope, the less chance of unauthorized access and the misusing of the functions and variables. |
| 1. Use Smart Pointers | In C++ raw pointers could lead to memory leaks and dangling pointers if they are not handled correctly. To prevent this from, use smart pointers as shared\_ptr or unique\_ptr that automatically manage memory leaks. |
| 1. Prefer standard library functions | C++ offers an extensive standard library that has been tested, optimized, is stable and well maintained over other libraries or self-created implementations. |
| 1. Regularly Patch and Update | Ensure that all compilers, libraries, and tools are patched and up to date, which will help protect the code from vulnerabilities. |
| 1. Use const correctly | Declare pointers and variables as const whenever possible to ensure that they can’t be modified or manipulated. |
| 1. Sanitize All Inputs | Validate and sanitize all input data before the data is used. This will help prevent attacks such as buffer overflows and SQL injections. |
| 1. Limit the use of dynamic memory | Dynamic memory allocation could lead to problems including memory leaks and heap corruption. Use automatic variables that get cleaned up when they go out of scope. |
| 1. Minimize the use of Global Variables | Global variables can be accessed and modified by any part of the code, which could lead to security vulnerabilities and/or unexpected behavior. When possible, use local variables and pass them as functions. |
| 1. Avoid buffer overflows | C/C++ does not perform automatic bound checking on arrays which could lead to buffer overflow vulnerabilities. Always ensure that the array size is checked against the index being accessed. To avoid buffer overflow, ensure that writing and reading is not past the end of the array. |

### 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-STR52-CPP | Use valid references, pointers, and iterators to reference elements of a basic\_string |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, data is invalidated after the call to replace(), and so its use in g() is undefined behavior. |
| #include <iostream>  #include <string>    extern void g(const char \*);    void f(std::string &exampleString) {  const char \*data = exampleString.data();  // ...  exampleString.replace(0, 2, "bb");  // ...  g(data);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the pointer to exampleString's internal buffer is not generated until after the modification from replace() has completed. |
| #include <iostream>  #include <string>    extern void g(const char \*);    void f(std::string &exampleString) {  // ...  exampleString.replace(0, 2, "bb");  // ...  g(exampleString.data());  } |

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

| **Principles(s):** Using an invalid reference, pointer, or iterator to a string object could allow an attacker to run arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2023.1 | DF4746, DF4747, DF4748, DF4749 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2023a | CERT C++: STR52-CPP | Checks for use of invalid string iterator (rule partially covered). |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-CTR53-CPP | Use valid iterator ranges |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to replace the initial character in the string with a capitalized equivalent. However, if the given string is empty, the behavior is undefined. |
| #include <string>  #include <locale>    void capitalize(std::string &s) {  std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to std::string::front() is made only if the string is not empty. |
| #include <string>  #include <locale>    void capitalize(std::string &s) {  if (s.empty()) {  return;  }    std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

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

| **Principles(s):** Using an invalid iterator range is similar to allowing a buffer overflow, which can lead to an attacker running arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| CodeSonar | 7.3p0 | LANG.MEM.BO | Buffer Overrun |
| Helix QAC | 2023.1 | C++3802 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |
| Polyspace Bug Finder | R2023a | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered). |
| PVS-Studio | 7.25 | V539, V662, V789 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-STR51-CPP | Do not attempt to create a std::string from a null pointer |

| **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):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 7.3p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2023.1 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2023.1 | 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 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-STR02-CPP | Sanitize data passed to complex subsystems |

| **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** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  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):** Failure to sanitize data passed to a complex subsystem can lead to an injection attack, data integrity issues, and a loss of sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 7.3p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Klocwork | 2023.2 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |
| Polyspace Bug Finder | R2023a | CERT C: Rec. STR02-C | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-MEM56-CPP | Do not store an already-owned pointer value in an unrelated smart pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the poly pointer value owned by a std::shared\_ptr object is cast to the D \* pointer type with dynamic\_cast in an attempt to obtain a std::shared\_ptr of the polymorphic derived type. However, this eventually results in undefined behavior as the same pointer is thereby stored in two different std::shared\_ptr objects. When g() exits, the pointer stored in derived is freed by the default deleter. Any further use of poly results in accessing freed memory. When f() exits, the same pointer stored in poly is destroyed, resulting in a double-free vulnerability. |
| #include <memory>    struct B {  virtual ~B() = default; // Polymorphic object  // ...  };  struct D : B {};    void g(std::shared\_ptr<D> derived);    void f() {  std::shared\_ptr<B> poly(new D);  // ...  g(std::shared\_ptr<D>(dynamic\_cast<D \*>(poly.get())));  // Any use of poly will now result in accessing freed memory.  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamic\_cast is replaced with a call to std::dynamic\_pointer\_cast(), which returns a std::shared\_ptr of the polymorphic type with the valid shared pointer value. When g() exits, the reference count to the underlying pointer is decremented by the destruction of derived, but because of the reference held by poly (within f()), the stored pointer value is still valid after g() returns. |
| #include <memory>    struct B {  virtual ~B() = default; // Polymorphic object  // ...  };  struct D : B {};    void g(std::shared\_ptr<D> derived);    void f() {  std::shared\_ptr<B> poly(new D);  // ...  g(std::dynamic\_pointer\_cast<D, B>(poly));  // poly is still referring to a valid pointer value.  } |

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

| **Principles(s):** Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM56 |  |
| Helix QAC | 2023.1 | DF4721, DF4722, DF4723 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| Polyspace Bug Finder | R2023a | CERT C++: MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |
| PVS-Studio | 7.25 | V1006 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-DCL03-CPP | Understand the termination behavior of assert() and abort() |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code that uses this structure to behave correctly. |
| struct timer {  uint8\_t MODE;  uint32\_t DATA;  uint32\_t COUNT;  };    int func(void) {  assert(offsetof(timer, DATA) == 4);  } |

| **Compliant Code** |
| --- |
| This compliant solution mimics the behavior of static\_assert in a portable manner. |
| #define JOIN(x, y) JOIN\_AGAIN(x, y)  #define JOIN\_AGAIN(x, y) x ## y    #define static\_assert(e) \  typedef char JOIN(assertion\_failed\_at\_line\_, \_\_LINE\_\_) [(e) ? 1 : -1]    struct timer {  uint8\_t MODE;  uint32\_t DATA;  uint32\_t COUNT;  };    static\_assert(offsetof(struct timer, DATA) == 4); |

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

| **Principles(s):** Static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities at compile time. The absence of static assertions, however, does not mean that code is incorrect. |
| --- |

**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 | Checked by clang-tidy |
| CodeSonar | 7.3p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-ERR50-CPP | Do not abruptly terminate the program |

| **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):** Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stdlib-use | Partially checked |
| CodeSonar | 7.3p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| Helix QAC | 2023.1 | C++5014 |  |
| Klocwork | 2023.1 | MISRA.TERMINATE  CERT.ERR.ABRUPT\_TERM |  |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced Enforcement |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR50-a  CERT\_CPP-ERR50-b  CERT\_CPP-ERR50-c  CERT\_CPP-ERR50-d  CERT\_CPP-ERR50-e  CERT\_CPP-ERR50-f  CERT\_CPP-ERR50-g  CERT\_CPP-ERR50-h  CERT\_CPP-ERR50-i  CERT\_CPP-ERR50-j  CERT\_CPP-ERR50-k  CERT\_CPP-ERR50-l  CERT\_CPP-ERR50-m  CERT\_CPP-ERR50-n | The execution of a function registered with 'std::atexit()' or 'std::at\_quick\_exit()' should not exit via an exception  Never allow an exception to be thrown from a destructor, deallocation, and swap  Do not throw from within destructor  There should be at least one exception handler to catch all otherwise unhandled exceptions  An empty throw (throw;) shall only be used in the compound-statement of a catch handler  Exceptions shall be raised only after start-up and before termination of the program  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point  Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)  Function called in global or namespace scope shall not throw unhandled exceptions  Always catch exceptions  Properly define exit handlers  The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  Avoid throwing exceptions from functions that are declared not to throw  The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |
| Polyspace Bug Finder | R2023a | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function (rule partially covered) |
| PVS-Studio | 7.25 | V667, V2014 |  |
| RuleChecker | 22.10 | stdlib-use | Partially checked |
| SonarQube C/C++ Plugin | 4.10 | S990 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Structures | STD-CTR51-CPP | Use valid references, pointers, and iterators to reference elements of a container. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, pos is invalidated after the first call to insert(), and subsequent loop iterations have undefined behavior. |
| #include <deque>    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** |
| --- |
| In this compliant solution, pos is assigned a valid iterator on each insertion, preventing undefined behavior. |
| #include <deque>    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):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 7.3p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2023.1 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Klocwork | 2023.1 | 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 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations | STD-DCL59-CPP | Do not define an unnamed namespace in a header file |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the variable v is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of v and then assigns a new value into it. However, because v is defined within an unnamed namespace, each translation unit operates on its own instance of v, resulting in unexpected output. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    namespace {  int v;  }    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  #include <iostream>    void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }    // b.cpp  #include "a.h"  #include <iostream>    void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }    int main() {  extern void f();  f(); // Prints v, sets it to 42  g(); // Prints v, sets it to 100  f();  g();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, v is defined in only one translation unit but is externally visible to all translation units, resulting in the expected behavior. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    extern int v;    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  #include <iostream>    int v; // Definition of global variable v    void f() {  std::cout << "f(): " << v << std::endl;  v = 42;  // ...  }    // b.cpp  #include "a.h"  #include <iostream>    void g() {  std::cout << "g(): " << v << std::endl;  v = 100;  }    int main() {  extern void f();  f(); // Prints v, sets it to 42  g(); // Prints v, sets it to 100  f(); // Prints v, sets it back to 42  g(); // Prints v, sets it back to 100  } |

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

| **Principles(s):** Defining an unnamed namespace within a header file can cause data integrity violations and performance problems but is unlikely to go unnoticed with sufficient testing. One-definition rule violations result in undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | unnamed-namespace-header | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL59 |  |
| Clang | 3.9 | cert-dcl59-cpp | Checked by clang-tidy |
| CodeSonar | 7.3p0 | LANG.STRUCT.DECL.ANH | Anonymous Namespace in Header File |
| Helix QAC | 2023.1 | C++2518 |  |
| Klocwork | 2023.1 | MISRA.NAMESPACE.UNMD |  |
| LDRA tool suite | 9.7.1 | 286 S, 512 S | Fully implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL59-a | There shall be no unnamed namespaces in header files |
| Polyspace Bug Finder | R2023a | CERT C++: DCL59-CPP | Checks for unnamed namespaces in header files (rule fully covered) |
| RuleChecker | 22.10 | unnamed-namespace-header | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | UnnamedNamespaceInHeader |  |
| PVS-Studio | 7.25 | V1068 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Strings | STD-STR53-CPP | Range check element access |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to replace the initial character in the string with a capitalized equivalent. However, if the given string is empty, the behavior is undefined. |
| #include <string>  #include <locale>    void capitalize(std::string &s) {  std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

| **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):** Unchecked element access can lead to out-of-bound reads and writes and write-anywhere exploits. These exploits can, in turn, lead to the execution of arbitrary code with the permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 7.3p0 | 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 | 2023.1 | C++3162, C++3163, C++3164, C++3165 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2023a | CERT C++: STR53-CPP | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |

### 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.

Ideally automation should be incorporated into every step of the DevSecOps process, to ensure the most secure system. During the plan stage, automation tools could help manage and organize the policy related tasks and requirements. During the creation stage, the code should have an automated static code analysis to detect any potential issues. In the testing phase, the tests should be automated to check for compliance and help identify vulnerabilities before the software is released. During the release phase, the deployment process should include automated integrated policy checks, so that if failed the deployment would fail. While the software is in operation, automated monitoring and alerting tools should be in place to keep track of any issues. Lastly monitoring the whole system with automated analysis tools and continually checking for vulnerabilities ensures the safety of the overall system.

### 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-STR52-CPP | High | Probable | High | P6 | L2 |
| STD-CTR53-CPP | High | Probable | High | P6 | L2 |
| STD-STR51-CPP | High | Likely | Medium | P18 | L1 |
| STD-STR02-CPP | High | Likely | Medium | P18 | L1 |
| STD-MEM56-CPP | High | Likely | Medium | P18 | L1 |
| STD-DCL03-CPP | Low | Unlikely | High | P1 | L3 |
| STD-ERR50-CPP | Low | Probable | Medium | P4 | L3 |
| STD-CTR51-CPP | High | Likely | Medium | P18 | L1 |
| STD-DCL59-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-STR53-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 at rest is designed to prevent the attacker from accessing the unencrypted data by ensuring the data is encrypted onto a hard drive, flash drive, or cloud storage. If the storage medium or system is attacked and compromised, encryption at rest ensures that the data remains inaccessible without the decryption keys. |
| Encryption at flight | Encryption at flight is the process of encrypting data that is being moved or transmitted between systems or over a network. Using encryption while the data is in transit ensures that the data can not be intercepted and read without the encryption keys. |
| Encryption in use | Encryption in use is data that is accessed or used by a user or an application. This is when the data is the most vulnerable, requiring the system to have good authentications in place to ensure the data is not compromised by an outside entity. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of the user, a device, or the system. To authenticate a user and allow access to the system the authentication process involves checking the credentials of the user. Some systems might have added security measures in place like two-factor authentication or biometrics. |
| Authorization | Authorization occurs after the user is authenticated, and determining what that user could do inside of the system. Every user will have permissions and rights determining what resources that user could access. |
| Accounting | Accounting is the process of keeping track and monitoring the resources the user accessed and what they did. Accounting collects a record of what databases were accessed and what the user did inside those databases. This provides a record of every user and could be used in the event of a compromise from within the company. |

**\***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.

Standard 1 – Principles 3,6

Standard 2 – Principles 2,6,10

Standard 3 – Principles 2,6,7,8,9

Standard 4 – Principles 4,7,8,9,10

Standard 5 – Principles 3,9

Standard 6 – Principles 4

Standard 7 – Principles – none of the ones I described.

Standard 8 – Principles 2, 3,4,6,7,8,9,10

Standard 9 – Principles 4,6,7,9

Standard 10 – Principles 2,4,6,7,9,10

**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 | 07/06/2023 | First Revision | James Soto |  |
| 1.1 | 07/20/2023 | Final Revision | James Soto |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

Sources

*Top 10 Secure Coding Practices - CERT Secure Coding - Confluence*. (n.d.). <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>