**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 | Validating input data means checking any data that is coming into a system to ensure that it contains values that are within the expectations of the program. This means ensuring numbers are within expected ranges as well as that the input does not contain anything that could be misinterpreted as code by the program. It’s also important to make sure the input does not exceed the bounds of the memory it will be stored in. |
| 1. Heed Compiler Warnings | Compiler warnings may not stop your program from compiling or running, but ignoring compiler warnings is not a good habit for developers. Often compiler warnings can alert you that there is something in your code that could be improved or something that may be potentially unsafe. Reviewing and understanding compiler warnings is essential to writing secure code. |
| 1. Architect and Design for Security Policies | It is important to design and write code in accordance with security policies and principles that are well defined prior to designing a new system or program. This can mean following and implementing your company’s security policies or following best practices or a personal set of security policies if you are an independent developer. |
| 1. Keep It Simple | Adding complexity to your code does not mean the same thing as adding security. The more complicated a program or security policy is, the tougher it is to design, implement, and maintain. Keep code as simple as possible while keeping security best practices and policies in mind. |
| 1. Default Deny | A program or system should deny access by default. This means that instead of automatically allowing everyone to use a system, it’s better to deny access by default and only allow access to those who should be able to access a program or system. This is especially important for areas of a program that contains sensitive data. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the principle of least privilege means only allowing someone enough access to complete a task and not providing them any further access to the system or program. For instance, if someone’s job is to run a report every week, they should only have access to the reporting functions that allow someone to print a report. They should not have full administrative access to the system to simply print a report. |
| 1. Sanitize Data Sent to Other Systems | Much like validating input data, it is important to validate data that is sent from one system to another. Sanitizing data before it’s sent to another system ensures that the correct data is sent and that it is formatted in a way that the other system expects and can make use of it. |
| 1. Practice Defense in Depth | Practicing defense in depth means not relying on a single security mechanism to secure an application. Defense in depts means using multiple layers of redundant security mechanisms in case one line of defense fails the program can still be protected by other layers of security. |
| 1. Use Effective Quality Assurance Techniques | By using effective quality assurance techniques, QA technicians can find problems early in the development process that can be fixed by the development team. A QA team that uses effective techniques knows which parts of the code are most important to test and the various kinds of tests that can be used to test a program’s functionality. Finding bugs before a program is released can prevent security issues from being found in the future. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard allows developers to all be on the same page when it comes to writing code that is secure. While no program is truly secure, if all programmers working on a project agree on and maintain secure coding practices, the likeliness of discovering an obvious security flaw due to bad program design or logic decreases, making the program more secure overall. |

### 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-DTS | Ensure that integer conversions do not result in lost or misinterpreted data. |

| **Noncompliant Code** |
| --- |
| This code will result in a loss of data because the signed char variable **sc** cannot store the large value held by the unsigned long integer **value**. |
| #include <limits>  void calculateSomething(void) {  unsigned long value = std::numeric\_limits<unsigned long>::max();  signed char sc;  sc = static\_cast<signed char>(value);  } |

| **Compliant Code** |
| --- |
| In this example of compliant code, the value of the variable **value** is checked to ensure its value is less than the maximum numeric limit of an unsigned char prior to attempting to store the value in the variable **sc**. |
| #include <limits>  void calculateSomething(void) {  unsigned long value = std::numeric\_limits<unsigned long>::max();  signed char sc;  if (value <= std::numeric\_limits<signed char>::max()) {  sc = static\_cast<signed char>(value);  } else {  // handle error  }  } |

| **Noncompliant Code** |
| --- |
| In this non-compliant example, the code does not validate ranges when converting from a signed to an unsigned type. |
| #include <limits>  void calculateSomething(signed si) {  // Cast eliminates compiler warning!  unsigned ui = static\_cast<unsigned>(si);  }  calculateSomething(std::numeric\_limits<int>::min()); |

| **Compliant Code** |
| --- |
| In this compliant example, |
| #include <limits>  void calculateSomething(signed si) {  unsigned ui;  if (si < 0) {  // handle error  } else {  ui = static\_cast<unsigned>(si); // cast eliminates warning.  }  calculateSomething(std::numeric\_limits<int>::min()); |

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

| **Principles(s):** 3) Architect and Design for Security Policies : Following best practices and keeping security in mind will cause developers to think logically about the kind of numbers they are storing and the potential values the variables can hold.  2) Heed Compiler Warnings: Without casing during the assignment, most compilers will warn programmers about the potential loss of data by the assignment operation. Casting only silences the warning but does not correct the underlying issue. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char. |
| CodeSonar | 7.3p0 | LANG.CAST.PC.AV | Cast: arithmetic type/void pointer |
| Astree | 22.04 | MISRA C:2012 Rules | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6, and 10.7 |
| LDRA tool suite | 9.7.1 | 93 S, 433 s, 434 S | Partially implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-DBZ | Ensure that division and remainder operations do not result in divide-by-zero errors. |

| **Noncompliant Code** |
| --- |
| This non-compliant code check to ensure that the operation on signed integers does not result in an overflow but does not attempt to prevent the operation from dividing by zero. |
| #include <limits>  void calculateSomething(signed long a, signed long b) {  signed long result;  if ((a == std::numeric\_limits<signed long>::max()) &&  (b == -1) {  // handle error }  } else {  result = a / b; }  } |

| **Compliant Code** |
| --- |
| The only modification is adding an OR expression which checks to ensure that the variable **b** is not equal to zero prior to attempting the division operation. |
| #include <limits>  void calculateSomething(signed long a, signed long b) {  signed long result;  if ((a == std::numeric\_limits<signed long>::max()) &&  (b == -1) || (b == 0) {  // handle error }  } else {  result = a / b; }  } |

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

| **Principles(s):** 3) Architect and Design for Security Policies – By following best coding practices and keeping security in mind, programs should be designed in a way that anticipates probably sources for errors such as attempting to divide by zero. By thinking about this potential while initially writing the code, these kinds of errors can be eliminated before they become runtime issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | int-division-by-zero | Fully checked |
| CodeSonar | 7.3p0 | LANG.ARITH.DIVZERO | Division by zero |
| Cppcheck | 1.66 | zerodiv  zerodivcond | Context sensitive analysis of division by zero. Not detected for division by struct member, array element, or pointer data that is 0. |
| Parasoft C/C++ test | 2022.2 | CERT\_C-INT33-a | Avoid division by zero |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-SCX | Do not pass a non-null-terminated character sequence to a library function that expects a string. |

| **Noncompliant Code** |
| --- |
| (This example uses features from the C++ 20 standard)  This non-compliant code example passes a non-null terminated c-style string to a function expecting a string. |
| #include <iostream>  void myFunction(void) {  char c\_str[3] = “cat”;  std::cout << std::format(“{}\n”, c\_str);  } |

| **Compliant Code** |
| --- |
| This compliant example does not specify the size of the array at declaration. The compiler will allocate the correct amount of memory including space for a null terminator. |
| #include <iostream>  void myFunction(void) {  char c\_str[] = “cat”;  std::cout << std::format(“{}\n”, c\_str);  } |

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

| **Principles(s):** 1) ValidateInput Data – While this is not an example of sanitizing untrusted data, it is still important to ensure that any data you pass to a function should is formatted properly so the receiving function works as intended.  4) Keep It Simple – This example shows that keeping it simple (by not specifying the size of the array during it’s declaration) avoids running into potential bugs and/or vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 73p0 | MISC.MEM.NTERM.CSTRING | Unterminated C String |
| Coverity | 2017.07 | STRING\_NULL | Fully implemented |
| TrustInSoft Analyzer | 1.38 | match format and arguments | Partially verified |
| LRDA tool suite | 9.7.1 | 404 S 600 S | Partially implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-SQL | Sanitize data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| This SQL statement has strings directly in the statement that could be used to inject malicious code. |
| std::string SQL = “ SELECT \* FROM USERS WHERE USERNAME = “ + username “ + “;”; |

| **Compliant Code** |
| --- |
| This code correctly parameterizes the arguments being used in the SQL statement. Functions provided by the SQL driver will allow the variables to be inserted safely. |
| std::string SQL = “SELECT & FROM USERS WHERE USERNAME = ?”;  // create prepared statement using functions from SQL driver… |

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

| **Principles(s):** 7) Sanitize Data Sent to Other Systems – Validating input data is important and, in this context, can prevent a SQL injection attack. It’s always important to verify data correctness prior to sending it to another system.  10) Adopt a Secure Coding Standard – Always follow best coding practices and keep security in mind throughout all phases of development. Knowing about various attacks such as SQL injection can help developers from making mistakes that potentially turn into security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Supported by stubbing/taint analysis |
| Klocwork | 2023.1 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |
| LDRA tool suit | 9.7.1 | 108 D, 109 D | Partially implemented |
| 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 |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-AFM | Do not access freed memory  Accessing memory that has been deallocated can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This non-compliant example stores a pointer to a temporary string object. This can result in undefined behavior when attempting to access a pointer to this temporary object. |
| #include <string>  using std::string;  string str\_func();  void display\_string(const char \*);  void myFunction() {  const char \*str = sr\_func().c\_str();  display\_string(str);  } |

| **Compliant Code** |
| --- |
| In this compliant example, the string objects are stored locally in a variable. Since these variables are not temporary values, it is safe to access their location in memory. |
| #include <string>  using std::string;  string str\_func();  void display\_string(const char \*s);  void myFunction() {  string str = str\_func();  const char \*cstr = str.c\_str();  display\_string(cstr);  } |

| **Noncompliant Code** |
| --- |
| In this non-compliant example, the dynamically allocated memory managed by the buff object is accessed beyond the point where the object’s destructor is called (implicitly). |
| #include <iostream>  #include <memory>  #include <cstring>  int main(int argc, const char \*argv[]) {  const char \*s = “”;  if (argc > 1) {  enum { BufferSize = 32 };  try {  std::unique\_ptr<char[]> buff(new char[BufferSize]);  std::memset(buff.get(), 0, BufferSize);  s = std:;strncpy(buff.get(), argv[1], BufferSize – 1);  } catch (std::bad\_alloc&) {  // handle error  }  }  std::cout << s << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the unique\_ptr is declared in the beginning of main()’s scope so it does not go out of scope prior to its usage. |
| #include <iostream>  #include <memory>  #include <cstring>  int main(int argc, const char \*argv[]) {  std::unique\_ptr<char[]> buff;  const char \*s = “”;  if (argc > 1) {  enum { BufferSize = 32 };  try {  buff.reset(new char[BufferSize]);  std::memset(buff.get(), 0, BufferSize);  s = std:;strncpy(buff.get(), argv[1], BufferSize – 1);  } catch (std::bad\_alloc&) {  // handle error  }  }  std::cout << s << std::endl;  return 0;  } |

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

| **Principles(s):** 3) Architect and Design for Security Policies – Designing with security in mind can prevent developers from making mistakes when managing memory dynamically. Using smart pointers can also help developers manage memory and prevent issues such as accessing freed memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use after free |
| LDRA tool suit | 9.7.1 | 483 S, 484 S | Partially implemented |
| Splint | 5.0 |  |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete | Checked by clang-tidy, but does not catch all violations of this rule. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-ASS | Understand the termination behavior or assert() and abort(). |

| **Noncompliant Code** |
| --- |
| The cleanup function will not run if the assert statement is triggered because it will call abort(). |
| void cleanup(void) {  // Delete temp files, restore states, etc..  }  int main (void) {  if (atexit(cleanup) != 0 ) {  // handle error  }  assert(//everything worked as expected);  } |

| **Compliant Code** |
| --- |
| This code asserts that the object is not null prior to inserting it onto the vector |
| void cleanup(void) {  // Delete temp files, restore states, etc..  }  int main (void) {  if (atexit(cleanup) != 0 ) {  // handle error  }  if (//something bad happened) {  exit(EXIT\_FAILURE);  }  } |

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

| **Principles(s):** 9) Use Effective Quality Assurance Techniques – Using consistent and effective QA techniques will help development teams from making mistakes that can cost developers and QA teams in terms of productivity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  | Can detect some violations of this rule but can only detect violations involving abort() since assert() is implemented as a macro. |
| LDRA tool suit | 9.7.1 | 44 S | Enhanced enforcement |
| Parasoft C/C++ test | 2022.2 | CERT\_C-ERR06-a | Do not use assertions |
| PC-lint Plus | 1.4 | 586 | Fully supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-HAX | Handle all exceptions. Failing to handle exceptions can lead to implementation defined behavior. |

| **Noncompliant Code** |
| --- |
| In this non-compliant example, neither f() or main() catch exceptions from throwing\_func(). |
| void throwing\_func() noexcept (false);  void f() {  throwing\_func();  }  int main(void) {  f();  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant code, a try/catch block is implemented in main() and can catch exceptions thrown by throwing\_func(). |
| void throwing\_func() noexcept (false);  void f() {  throwing\_func();  }  int main(void) {  try {  f();  } catch (…){  // Handle error  }  return 0;  } |

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

| **Principles(s):** 3) Architect and Design for Security Policies & Adopt a Secure Coding Standard – Following best practices and security policies will lead developers to write better code that is capable of handling errors gracefully rather than allowing errors to crash the program at runtime. |
| --- |

**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 | Partially checked |
| CodeSonar | 7.3p0 | CertC++-ERR51 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-ERR51-a | Always catch exceptions |
| RuleChecker | 22.10 | main-function-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Valid References | STD-008-SIB | Use valid references, pointers, and iterators to reference elements of a container. |

| **Noncompliant Code** |
| --- |
| In this non-compliant example **pos** is no longer valid after the first call to insert since the first insertion occurs at the beginning of the deque. |
| #include <deque>  void myFunction(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 the following example, the iterator is updated after the insert statement so the iterator doesn’t become invalid after updating the vector. |
| #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);  }  } |

| **Noncompliant Code** |
| --- |
| In this non-compliant example **pos** is no longer valid after the first call to insert since the first insertion occurs at the beginning of the vector. |
| #include <vector>  void myFunction(const double \*items, std::size\_t count) {  std::vector<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 non-compliant example **pos** is no longer valid after the first call to insert since the first insertion occurs at the beginning of the vector. |
| #include <vector>  void myFunction(const double \*items, std::size\_t count) {  std::vector<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):** 10) Adopt a Secure Coding Standard – Following best practices can help developers from making common mistakes such as working with an invalidated iterator. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | overflow\_upon\_dereference |  |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use after free |
| Klocwork | 2023.1 | ITER.CONTAINER.MODIFIED |  |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Allocation | STD-009-MAE | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| In this example, the result of the allocation of space for the copy array is not checked. |
| #include <cstring>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  delete [] copy;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the value of copy is checked to ensure no error occurred during memory allocation prior to attempting to populate the array. |
| #include <cstring>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  delete [] copy;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the std::bad\_alloc exception is caught and handled in the case where memory allocation fails, preventing the memcopy() function call after memory allocation was not successful. |
| #include <cstring>  #include <new>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy;  try {  copy = new int[size];  } catch (std::bad\_alloc) {  // handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  delete [] copy;  } |

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

| **Principles(s):** 9) Use Effective Quality Assurance Techniques – Using effective QA techniques such as handling errors gracefully will make life easier for the QA team and prevent potential issues related to dynamically allocating memory.  10) Defense-in-Depth – By using multiple layers of security, developers can help prevent issues in the case that one security-mechanism fails by having redundant layers of security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled. |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| Polyspace Bug Finder | R2023a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| PVS-Studio | 7.2.4 | V522, v668 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Constructor Canonical Order | STD-010-CCO | Write constructor member initializers in the canonical order  Initializing members out of order can lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| This non-compliant example initializes the class fields out of the order in which they were defined. |
| class C {  int dependsOnSomeVal;  int someVal;  public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| In the compliant example, the order in which the class members are defined has been ordered properly. |
| class C {  int someVal;  int dependsOnSomeVal;  public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** 4) Keep It Simple – Keep things simple by making sure class members are declared in an order that makes sense and ensuring that member initialization occurs in the same order to prevent the program from exhibiting undefined behaviors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | initializer-list-order | Fully checked |
| Clang | 3.9 | -Wreorder |  |
| Klocwork | 2023.1 | CERT.OOP.CTOR.INIT\_ORDER |  |
| RuleChecker | 22.10 | initializer-list-order | Fully checked |

### 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.

The existing DevOps process can be modified to automate enforcements of the standards in the security policy by ensuring that developers and build systems have the tools and plugins needed to ensure developers are following best practices. Static analysis tools can be used by developers and through the continuous integration systems used to check code when it has been submitted to prevent common mistakes. The security team should be involved in the design phase in order to ensure that security is built into the system and not an afterthought. Tools like static analysis can be used during the build and verification phases to catch common issues that developers may have overlooked. Security audits should be conducted and reviewed during the production phases such the transition and health check phase, the monitoring and detecting phase, in addition to reviewing logs after maintenance has been completed. This will allow the security team to respond properly in the event of a security breech. When a system is in production, it’s essential to keep good logs and monitor those logs for any signs of security violations so the team can take proper actions to resolve any security related concerns.

### 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-DTS | High | Medium | High | High | 2 |
| STD-002-DBZ | Low | High | Medium | Medium | 2 |
| STD-003-NTC | High | High | Medium | High | 4 |
| STD-004-SQL | High | High | Medium | High | 5 |
| STD-005-AFM | High | High | Medium | High | 5 |
| STD-006-ASS | Medium | Low | Medium | Low | 1 |
| STD-007-HAX | Low | High | Medium | Medium | 2 |
| STD-008-UVR | High | Medium | High | High | 4 |
| STD-009-MAE | High | Medium | Medium | High | 4 |
| STD-010-CCO | Medium | Low | Medium | Low | 2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption of data in rest protects data that is being stored such as data being stored on a hard disk. The policy applies because data at rest that is unencrypted can easily be read if the data is exposed/leaked. All data at rest should be secured using industry standard encryption. |
| Encryption at flight | Data is vulnerable when it is in transit. Just as you wouldn’t send a letter with confidential information in a non-security envelope, data should never be transmitted if it is not encrypted. All data being transmitted should be encrypted using industry standard encryption algorithms/ciphers. |
| Encryption in use | Even while being used by an application, data is vulnerable. A program running on a computer stores data within the computer’s memory. All sensitive data being used by an application should encrypt the data prior to storing it in memory to prevent malware from obtaining potentially sensitive data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying who or what an entity is. When you log on using a username and password, you are essentially verifying who you say you are. All users of company systems should be authenticated first. |
| Authorization | Authorization is determining whether an entity is allowed or disallowed from using a specified resource. For example, a data entry employee does not need permission to create and delete tables in an SQL database since they are just inputting data into existing databases. This policy prevents users from accessing functions that they should not have access to and protects company systems in the event of a user account being compromised. The principle of least privilege should be used to ensure users do not have more access than they need to complete their job. |
| Accounting | Accounting is the process of documenting things related to security such as authentication, and authorization. Logs should be kept that indicate when user’s authentication, fail to authenticate, or attempt to access resources they are not allowed to access. This policy allows the security team to monitor systems to make sure they are secure and that they can determine when a security incident has occurred. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 03/20/2023 | Milestone One Completed | Charles Haines | pending approval |
| 1.2 | 04/07/2023 | Final draft completed | Charles Haines | pending approval |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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