**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input validation involves making sure attackers cannot input any malicious phrasing or files. For example, making sure “word=word” or “digit=digit (1=1)” is impossible to input. Untrusted sources must all be input validated. |
| 1. Heed Compiler Warnings | Compiler warnings must all be fixed since these indicate that something is not correct within the program. If the serious problems are not fixed, attackers can use this to their advantage. A programmer should compile code to the highest level of warnings available so that there is no stone left unturned when it comes to security. |
| 1. Architect and Design for Security Policies | A security policy is a set of rules on a system that protects sensitive resources. Creating a software architecture that is designed to enforce security policies will be beneficial for a system. |
| 1. Keep It Simple | Keeping a program simple will help a developer to fix errors quickly and make it possible for less errors to happen. If a program becomes too elaborate, it could make it harder to fix vulnerabilities quickly. |
| 1. Default Deny | Permissions should be defaulted to deny for any untrusted users. But there can be allowed permissions for trusted people like administrators for example. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege is a rule of giving a user the least amount of permission possible. Practicing this principle is necessary for security since if we give every user the highest permissions, we will easily have those who would abuse it and attack. |
| 1. Sanitize Data Sent to Other Systems | To sanitize data that is sent to other systems, this involves making sure all data that is sent is pure and does not contain anything malicious. It is important to sanitize the data before sending to make sure it conforms to each system’s “rules.” If it doesn’t conform, has invalid characters for example, it could make the systems potentially vulnerable to attacks. |
| 1. Practice Defense in Depth | Defense in Depth (DiD) is important to follow since it provides layers of security and makes it harder for hackers to attack. For example, a company may want to add a firewall to each computer the employees work on, then make sure passwords are a certain length and special characters, and then add two-step authentication to each account they set up for customers and employees. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance is important to practice with every program made. Testing a program thoroughly can help prevent and eliminate fatal errors. Techniques like fuzz testing and penetration testing can help immensely. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard can help with keeping a programmer on track with proper security techniques. If a standard is set, it’s easier for someone to follow and not miss important security techniques. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | Do not cast to an out-of-range enumeration value |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Enumerations in C++ come in two forms: *scoped* enumerations in which the underlying type is fixed and *unscoped* enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. |

| **Noncompliant Code** |
| --- |
| This piece of code attempts to check whether a value is within the range of acceptable enumeration values. When casted, it may not be able to represent the integer value given. |
| **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 code correctly checks that the value can be represented by the enumeration type before conversion. |
| **enum** EnumType {  First,  Second,  Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

| **Principles(s):** The principle that would fit the most with this standard would be number 1. It fits this principle since the value needs to be validated so that it is not being cast to an out-of-range enumeration value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | cast-integer-to-enum | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | --------------- |
| CodeSonar | 7.3p0 | LANG.CAST.COARCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2023.1 | C++3013 | --------------- |

#### Coding Standard 2

| **Coding Standard** | **Label** | Use valid references, pointers, and iterators to reference elements of a container |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)]. Pointers, references, and iterators share a close relationship in which it is required that referencing values be done through a valid iterator, pointer, or reference. Storing an iterator, reference, or pointer to an element within a container for any length of time comes with a risk that the underlying container may be modified such that the stored iterator, pointer, or reference becomes invalid. For instance, when a sequence container such as std::vector requires an underlying reallocation, outstanding iterators, pointers, and references will be invalidated [[Kalev 99](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-Kalev99)]. Use only a [valid pointer](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-validpointer), reference, or iterator to refer to an element of a container. |

| **Noncompliant Code** |
| --- |
| POS is invalidated after the first call to insert(). The loop iterators 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** |
| --- |
| POS is now assigned to a valid iterator on each insert() call. |
| #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);  }  } |

| **Principles(s):** The principle this would fit with is number 1. Validating that proper references, pointers, and iterators are being used is important in keeping attackers from exploiting them. |
| --- |

**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 |
| Helix QAC | 2023.1 | DF4746, DF4747, DF4748, DF4749 | ---------------- |
| Klocwork | 2023.1 | ITER.CONTAINER.MODIFIED | ---------------- |

#### Coding Standard 3

| **Coding Standard** | **Label** | Do not attempt to create a std::string from a null pointer |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The std::basic\_string type uses the *traits* design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Specifically, the std::basic\_string class is paired with std::char\_traits to create the std::string, std::wstring, std::u16string, and std::u32string classes. The std::char\_traits class is explicitly specialized to provide policy-based implementation details to the std::basic\_string type. One such implementation detail is the std::char\_traits::length() function, which is frequently used to determine the number of characters in a null-terminated string. According to the C++ Standard, [char.traits.require], Table 62 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], passing a null pointer to this function is [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior) because it would result in dereferencing a null pointer. |

| **Noncompliant Code** |
| --- |
| A std::string object is created from the results of a call to std::getenv(). Since the std::getenv() returns a null pointer on failure, it can lead to undefined behavior when the environment variable does not exist. |
| #include <cstdlib>  #include <string>    **void** f() {  std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| 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()) {  // ...  }  } |

| **Principles(s):** The principle that works here is number 1 and 10. Number 1 fits here since it is our job to ensure the string is correct and that we do not create one from a pointer. Number 10 fits also since adopting a policy on “not creating std::string’s from a null pointer” will help the team remember to watch out for it throughout the process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Assert\_failure | --------------- |
| CodeSonar | 7.3p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2023.1 | DF4770, DF4771, DF4772, DF4773, DF4774 | ---------------- |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-STR51-a | Avoid null pointer derefencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | Exclude user input from format strings |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C] | Never call a formatted I/O function with a format string containing a [tainted value](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-taintedvalue) . An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [[Seacord 2013b](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Seacord2013)]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the %n conversion specifier. This applied to both the C Standard and C++ libraries. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of [untrusted data](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-untrusteddata) that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;  /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {  /\* Handle error \*/  } **else** **if** (ret >= len) {  /\* Handle truncated output \*/  }  **fprintf**(stderr, msg);  **free**(msg); |

| **Compliant Code** |
| --- |
| This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;  /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {  /\* Handle error \*/  } **else** **if** (ret >= len) {  /\* Handle truncated output \*/  }  **fputs**(msg, stderr);  **free**(msg);  } |

| **Principles(s):** The principles that fit here are numbers 1, 6,9, and 10. Number 1 has a lot to do with validating input data when it comes to SQL injections. Formatted strings should not include a tainted value and there shouldn’t be a function used that will output the string and the issue it produces. Number 6 should be applied where certain users cannot get to see the input string error message and that means they need the least number of privileges. Number 9 should be practiced preventing an SQL injection also by making sure the quality of the code is good. Number 10 means that there should be a principle in place for the team to remember how to prevent SQL injections from happening. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 7.3p0 | IO.INJ.FMT MISC.FMT | Format string injection Format string |
| Parasoft C/C++ test | 2022.2 | CERT\_C-FIO30-a CERT\_C-FIO30-b CERT\_C-FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable Never use unfiltered data from an untrusted user as the format parameter |

#### Coding Standard 5

| **Coding Standard** | **Label** | Do not access freed memory |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-vulnerability).  It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {  S \*s = **new** S;  // ...  **delete** s;  // ...  s->f();  } |

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

| **Principles(s):** Not accessing freed memory falls under principles 2 and 10. Number 2 fits perfectly here since a compiler warning will most likely come up if there are any memory deallocation issues. Number 10 fits because if the team is reminded of how memory allocation and deallocation work, we can avoid anyone from gaining access through improper memory handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | **dangling\_pointer\_use** | ------------------------------------- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-MEM50** | -------------------------------------- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 7.3p0 | **ALLOC.UAF** | Use after free |

#### Coding Standard 6

| **Coding Standard** | **Label** | Incorporate diagnostic tests using assertions |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Incorporate diagnostic tests into your program using, for example, the assert() macro.  The assert macro expands to a void expression:   |  | | --- | | #include <**assert**.h>  **void** **assert**(scalar expression); |   When it is executed, if expression (which must have a scalar type) is false, the assert macro outputs information about the failed assertion (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function) on the standard error stream, in an [implementation-defined](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-implementation-definedbehavior) format, and calls the abort() function.  This standard was from the C Standard Library but also applies to the C++ library. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. |
| **char** \*dupstring(**const** **char** \*c\_str) {  **size\_t** len;  **char** \*dup;    len = **strlen**(c\_str);  dup = (**char** \*)**malloc**(len + 1);  **assert**(NULL != dup);    **memcpy**(dup, c\_str, len + 1);  **return** dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion. |
| **char** \*dupstring(**const** **char** \*c\_str) {  **size\_t** len;  **char** \*dup;    len = **strlen**(c\_str);  dup = (**char**\*)**malloc**(len + 1);  /\* Detect and handle memory allocation error \*/  **if** (NULL == dup) {  **return** NULL;  }    **memcpy**(dup, c\_str, len + 1);  **return** dup;  } |

| **Principles(s):** The principles that work for assertions are number 2 and 9. Number 2 will be important for heeding any compiler warnings that will pop up from assertions. Number 9 corresponds here since assertions are used for testing if inputs are valid or not. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2022.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | Handle all exceptions |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  If no matching handler is found, the function std::terminate() is called; whether or not the stack is unwound before this call to std::terminate() is implementation-defined.  The default terminate handler called by std::terminate() calls std::abort(), which [abnormally terminates](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination) the process. When std::abort() is called, or if the [implementation](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-implementation) does not unwind the stack prior to calling std::terminate(), destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for [denial-of-service](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-denial-of-service) attacks. For more information on implicitly calling std::terminate(), see [ERR50-CPP. Do not abruptly terminate the program](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR50-CPP.+Do+not+abruptly+terminate+the+program). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {  throwing\_func();  }    **int** main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {  throwing\_func();  }    **int** main() {  **try** {  f();  } **catch** (...) {  // Handle error  }  } |

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

| **Principles(s):** Principles 2 and 4 apply with exceptions. The compiler will mostly tell a programmer if there are any exceptions that happen and what happened. Also keeping the application simple can help with making less exceptions errors happen. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | main-function-catch-all early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 | ------------------------------------------- |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2023.1 | C++4035, C++4036, C++4037 | -------------------------------------------- |

#### Coding Standard 8

| **Coding Standard** | **Label** | A signal handler must be a plain old function |
| --- | --- | --- |
| **Signal Handlers** | [STD-008-CPP] | The C++14 Standard, [support.runtime], paragraph 10 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  The common subset of the C and C++ languages consists of all declarations, definitions, and expressions that may appear in a well-formed C++ program and also in a conforming C program. A POF (“plain old function”) is a function that uses only features from this common subset, and that does not directly or indirectly use any function that is not a POF, except that it may use plain lock-free atomic operations. A plain lock-free atomic operation is an invocation of a function f from Clause 29, such that f is not a member function, and either f is the function atomic\_is\_lock\_free, or for every atomic argument A passed to f, atomic\_is\_lock\_free(A) yields true. All signal handlers shall have C linkage. The behavior of any function other than a POF used as a signal handler in a C++ program is implementation-defined.228  Footnote 228 states the following:  In particular, a signal handler using exception handling is very likely to have problems. Also, invoking std::exit may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler.  If your signal handler is not a plain old function, then the behavior of a call to it in response to a signal is [implementation-defined](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-implementation-definedbehavior), at best, and is likely to result in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). All signal handlers must meet the definition of a plain old function. In addition to the restrictions placed on signal handlers in a C program, this definition also prohibits the use of features that exist in C++ but not in C (such as non-POD [non–plain old data] objects and exceptions). This includes indirect use of such features through function calls. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the signal handler is declared as a static function. However, since all signal handler functions must have C language linkage, and C++ is the default language linkage for functions in C++, calling the signal handler results in undefined behavior. |
| #include <csignal>    **static** **void** sig\_handler(**int** sig) {  // Implementation details elided.  }    **void** install\_signal\_handler() {  **if** (SIG\_ERR == std::**signal**(SIGTERM, sig\_handler)) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution defines sig\_handler() as having C language linkage. As a consequence of declaring the signal handler with C language linkage, the signal handler will have external linkage rather than internal linkage. |
| #include <csignal>    **extern** "C" **void** sig\_handler(**int** sig) {  // Implementation details elided.  }    **void** install\_signal\_handler() {  **if** (SIG\_ERR == std::**signal**(SIGTERM, sig\_handler)) {  // Handle error  }  } |

| **Principles(s):** Principles 2, 3, and 9 work for signal handlers. The compiler will most likely tell the programmer if something is amiss with the handler. Setting up a well-rounded architecture for security policies can help with reminding programmers of when and where to use signal handlers along with properly using them. Also having a good quality assurance tester can help figure out faster where the signal is getting caught up. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.1 | C++2888 | ------------------------------------------ |
| Klocwork | 2023.1 | CERT.MSC.SIG\_HANDLER.POF | ------------------------------------------ |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-MSC54-a | Properly define signal handler |

#### Coding Standard 9

| **Coding Standard** | **Label** | Avoid deadlock by locking in a predefined order |
| --- | --- | --- |
| **Concurrency** | [STD-009-CPP] | Mutexes are used to prevent multiple threads from causing a [data race](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-datarace) by accessing the same shared resource at the same time. Sometimes, when locking mutexes, multiple threads hold each other's lock, and the program consequently [deadlocks](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-deadlock). Four conditions are required for deadlock to occur:   * mutual exclusion (At least one nonshareable resource must be held.), * hold and wait (A thread must hold a resource while awaiting availability of another resource.), * no preemption (Resources cannot be taken away from a thread while they are in-use.), and * circular wait (A thread must await a resource held by another thread which is, in turn, awaiting a resource held by the first thread.).   Deadlock needs all four conditions, so preventing deadlock requires preventing any one of the four conditions. One simple solution is to lock the mutexes in a predefined order, which prevents circular wait. |

| **Noncompliant Code** |
| --- |
| The behavior of this noncompliant code example depends on the runtime environment and the platform's scheduler. The program is susceptible to deadlock if thread thr1 attempts to lock ba2's mutex at the same time thread thr2 attempts to lock ba1's mutex in the deposit() function. |
| #include <mutex>  #include <thread>    **class** BankAccount {  **int** balance;  **public**:  std::mutex balanceMutex;  BankAccount() = **delete**;  **explicit** BankAccount(**int** initialAmount) : balance(initialAmount) {}  **int** get\_balance() **const** { **return** balance; }  **void** set\_balance(**int** amount) { balance = amount; }  };    **int** deposit(BankAccount \*from, BankAccount \*to, **int** amount) {  std::lock\_guard<std::mutex> from\_lock(from->balanceMutex);    // Not enough balance to transfer.  **if** (from->get\_balance() < amount) {  **return** -1; // Indicate error  }  std::lock\_guard<std::mutex> to\_lock(to->balanceMutex);    from->set\_balance(from->get\_balance() - amount);  to->set\_balance(to->get\_balance() + amount);    **return** 0;  }    **void** f(BankAccount \*ba1, BankAccount \*ba2) {  // Perform the deposits.  std::**thread** thr1(deposit, ba1, ba2, 100);  std::**thread** thr2(deposit, ba2, ba1, 100);  thr1.join();  thr2.join();  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates the circular wait condition by establishing a predefined order for locking in the deposit() function. Each thread will lock on the basis of the BankAccount ID, which is set when the BankAccount object is initialized. |
| #include <atomic>  #include <mutex>  #include <thread>    **class** BankAccount {  **static** std::atomic<unsigned **int**> globalId;  **const** unsigned **int** id;  **int** balance;  **public**:  std::mutex balanceMutex;  BankAccount() = **delete**;  **explicit** BankAccount(**int** initialAmount) : id(globalId++), balance(initialAmount) {}  unsigned **int** get\_id() **const** { **return** id; }  **int** get\_balance() **const** { **return** balance; }  **void** set\_balance(**int** amount) { balance = amount; }  };    std::atomic<unsigned **int**> BankAccount::globalId(1);    **int** deposit(BankAccount \*from, BankAccount \*to, **int** amount) {  std::mutex \*first;  std::mutex \*second;    **if** (from->get\_id() == to->get\_id()) {  **return** -1; // Indicate error  }    // Ensure proper ordering for locking.  **if** (from->get\_id() < to->get\_id()) {  first = &from->balanceMutex;  second = &to->balanceMutex;  } **else** {  first = &to->balanceMutex;  second = &from->balanceMutex;  }  std::lock\_guard<std::mutex> firstLock(\*first);  std::lock\_guard<std::mutex> secondLock(\*second);    // Check for enough balance to transfer.  **if** (from->get\_balance() >= amount) {  from->set\_balance(from->get\_balance() - amount);  to->set\_balance(to->get\_balance() + amount);  **return** 0;  }  **return** -1;  }    **void** f(BankAccount \*ba1, BankAccount \*ba2) {  // Perform the deposits.  std::**thread** thr1(deposit, ba1, ba2, 100);  std::**thread** thr2(deposit, ba2, ba1, 100);  thr1.join();  thr2.join();  } |

| **Principles(s):** Principles 2, 4, and 10 work for concurrency. The compiler will tell the programmer if something is wrong with any of the threads usually, so it is important to pay attention to those first. When using threads, it is important to keep the application as simple as possible or it can be confusing on where the thread/s are deadlocking. Also adopting a good policy for reminding the team what to do for deadlocking threads can be a good place to start for avoiding fatal concurrency errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | CONCURRENCY.LOCK.ORDER | Conflicting lock order |
| Coverity | 6.5 | DEADLOCK | Fully implemented |
| Helix QAC | 2023.1 | C++1772, C++1773 |  |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-CON53-a | Do not acquire locks in different order |

#### Coding Standard 10

| **Coding Standard** | **Label** | Write constructor member initializers in the canonical order |
| --- | --- | --- |
| **Constructors** | [STD-010-CPP] | The member initializer list for a class constructor allows members to be initialized to specified values and for base class constructors to be called with specific arguments. However, the order in which initialization occurs is fixed and does not depend on the order written in the member initializer list. The C++ Standard, [class.base.init], paragraph 11 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  In a non-delegating constructor, initialization proceeds in the following order: — First, and only for the constructor of the most derived class, virtual base classes are initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where “left-to-right” is the order of appearance of the base classes in the derived class base-specifier-list. — Then, direct base classes are initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the mem-initializers). — Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers). — Finally, the compound-statement of the constructor body is executed. [Note: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. —end note]  Consequently, the order in which member initializers appear in the member initializer list is irrelevant. The order in which members are initialized, including base class initialization, is determined by the declaration order of the class member variables or the base class specifier list. Writing member initializers other than in canonical order can result in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior), such as reading uninitialized memory.  Always write member initializers in a constructor in the canonical order: first, direct base classes in the order in which they appear in the *base-specifier-list* for the class, then nonstatic data members in the order in which they are declared in the class definition. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the member initializer list for C::C() attempts to initialize someVal first and then to initialize dependsOnSomeVal to a value dependent on someVal. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of someVal results in an [unspecified value](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-unspecifiedvalue) being stored into dependsOnSomeVal. |
| **class** C {  **int** dependsOnSomeVal;  **int** someVal;    **public**:  C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| This compliant solution changes the declaration order of the class member variables so that the dependency can be ordered properly in the constructor's member initializer list. |
| **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):** The principles that work here are numbers 3, 4, and 10. A team should design a good security policy for constructors since it’s easy to forget the simplest rules for how constructors should be structured. Keeping the application simple can help by not having too many constructors to confuse programmers. Also adopting this security policy will potentially help keep the program concise and not confusing for the team. |
| --- |

**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 |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP53 |  |
| Clang | 3.9 | -Wreorder |  |
| CodeSonar | 7.3p0 | LANG.STRUCT.INIT.OOMI | Out of Order Member Initializers |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



### Automation



Automation belongs in every stage of the DevSecOps diagram. Detecting security flaws early helps with preventing more issues later. Though automation will help the best during the building process and verify and test sections since this is where the most mistakes will happen with code, it’s still always a good idea to practice using it at every stage. Automation tools help the team to see what they’re doing wrong and how to mitigate those issues safely. Always practice defense in depth (DiD) at every stage also since it will help create fewer problems that are unseen.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Unlikely | Medium | 4 | 3 |
| STD-002-CPP | High | Probable | High | 6 | 2 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-C | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-C | Low | Unlikely | High | 1 | 3 |
| STD-007-CPP | Low | Probable | Medium | 4 | 3 |
| STD-008-CPP | High | Probable | High | 6 | 2 |
| STD-009-CPP | Low | Probable | Medium | 4 | 3 |
| STD-010-CPP | Medium | Unlikely | Medium | 4 | 3 |

### Create Policies for Encryption and Triple A

| **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is for ensuring something that is stored, for example on a disk or hard drive, is encrypted and cannot be accessed by an attacker. The data that is not moving from device to device must be encrypted to ensure that no one unauthorized can get to it. The policy that should be implemented is to always encrypt data even if it’s resting in storage. |
| Encryption at flight | Encryption in flight is referring to encrypting data that is “on the move”. Data that is being should be encrypted also since an unauthorized person could intercept the data being transmitted and take it for personal gain. The policy is to encrypt data that is being transmitted. |
| Encryption in use | Encryption in use is encrypting data that is currently in use, for example, if a user is opening a file to read. Certain permissions should be set for certain data depending on how important that data is. The policy is to encrypt data that is being accessed. |

| **Triple-A Framework** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user to make sure they are who they say they are. Login-in usernames and passwords are prime examples of an authentication process. 2-step authentication is a newer way of authenticating the user but with more steps needed so it makes the account safer from attackers. The policy to apply here is to always have a user login that requires a specific amount of character and special characters. 2-step authentication should be used also on all accounts. |
| Authorization | Authorization is where the privileges of the user are determined. For example, the user can read a file but cannot write or delete the file from the database or vice versa. Roles can help with setting up different permissions for different types of users. For this system, users should have different levels of what can be assessed depending on their role. Administrators get a higher level of permissions than a plain user. |
| Accounting | Accounting is the process of logging action performed by users. For example, logging what files were accessed in the database, what actions were performed, and if there were changes to the database. For the policy, there should be an active log of when and what was accessed at all times and by whom. |

### 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

## 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 | 06/07/2023 | Added New Sections to Template | Alexis Indick |  |
| 1.2 | 06/09/2023 | Final Template Revision | Alexis Indick |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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