**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 | Input needs to be accurate and trusted. Validation ensures data is both correct and secure otherwise it should not be used. |
| 1. Heed Compiler Warnings | These warnings are useful to alert developers of potential errors or bugs in the code. Errors prevent code from compiling, warnings or other bugs might not prevent compilation but still pose problems. |
| 1. Architect and Design for Security Policies | Have security in mind when designing and planning the software architecture. Separating systems into sub systems with various authorization levels for example. |
| 1. Keep It Simple | The more complicated code gets, and design overall, the likelihood of errors or issues can increase. Keeping things simple and easy to digest can minimize problems and make security more straightforward. |
| 1. Default Deny | Access denial being the default is a good general practice. Permitting access based on conditions of security/protection is a safe principle to follow. |
| 1. Adhere to the Principle of Least Privilege | Executing processes with the minimal required privileges and only escalating privileges when absolutely needed for the fewest time possible is a great practice that improves overall security by making it harder for attackers to execute commands. |
| 1. Sanitize Data Sent to Other Systems | If data is not sanitized effectively, the risk of SQL injections opens up. Unused functions or function calls made out of context can pass and open up vulnerabilities. |
| 1. Practice Defense in Depth | Multiple layers of defense in order to cover a variety of things is better than focusing on only one aspect of defense. |
| 1. Use Effective Quality Assurance Techniques | Testing code thoroughly and rigorously, making sure multiple sets of eyes or audits go over it helps greatly. Fuzz and penetration testing, various QA programs or systems help identify any issues and corrections can be made. |
| 1. Adopt a Secure Coding Standard | From the beginning, applying coding standards in the language and platform used to setup a secure foundation. |

### 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** | [DCL60-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule) | Obey the one-definition rule |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two different translation units define a class of the same name with differing definitions. Although the two definitions are functionally equivalent (they both define a class named S with a single, public, nonstatic data member int a), they are not defined using the same sequence of tokens. This code example violates the ODR and results in undefined behavior. |
| // a.cpp  **struct** S {  **int** a;  };    // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| The correct mitigation depends on programmer intent. If the programmer intends for the same class definition to be visible in both translation units because of common usage, the solution is to use a header file to introduce the object into both translation units, as shown in this compliant solution |
| // S.h  **struct** S {  **int** a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** Adopt a secure coding standard- maintaining a single definition for an object across all translation units do that linking will behave deterministically. This supports how nontrivial C++ programs are often divided into multiple translation units that are later linked together to form an executable.. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Type-compatibility | Partially checked |
| CodeSonar | 7.4p0 | LANG.STRUCT.DEF.FDH | Function defined in header file |
| LDRA tool suite | 9.8.1 | 286s, 287s | Fully implemented |
| Polyspace Bug Finder | R2023b | CERT C++: DCL60-CPP | Checks for inline constraints not respected(rule partially covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | EXP53-CPP | Do not read uninitialized memory |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    **void** f() {  **int** i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>    **void** f() {  **int** i = 0;  std::cout << i;  } |

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

| **Principles(s):** Keep it simple- clean code and proper object initialization ensures that unitialized memory is not being read |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Uninitialized read | Partially checked |
| Helix QAC | 2023.2 | C++2728 | Checks |
| LDRA tool suite | 9.8.1 | 286s, 287s | Fully implemented |
| Polyspace Bug Finder | R2023b | CERT C++: EXP53-CPP | Checks for non-initialized pointers and variables |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    **void** f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Adopt a secure coding standard- using std::string isntead of bounded arrays is the compliant solution, unbounded inputs could lead to buffer overflows, secure coding mitigates this |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | P6 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.8.1 | 157s | Partially implemented |
| Polyspace bug finder | R2023b | Cert\_C STR50 | Rule covered |
| Astree | 22.10 | CERT-CPP.STR.50  check\_stream\_input\_char\_array | https://www.absint.com/releasenotes/astree/22.10/index.htm#rulechecker |
| ParasoftC/C++test | 2023.1 | CertSTR50 | Partially supported |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | EXP54-CPP | Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in undefined behavior. |
| **struct** S {  **void** mem\_fn();  };    **void** f() {  S \*s;  s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, storage is obtained for the pointer prior to calling S::mem\_fn(). |
| **struct** S {  **void** mem\_fn();  };    **void** f() {  S \*s = **new** S;  s->mem\_fn();  **delete** s;  } |

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

| **Principles(s):** Sanitize data- sanitizing data is an effective way to prevent SQL injections, especially when said data is used by different systems it becomes vital that it is secure |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klockwork | 2021.4 | UNINIT.HEAP.MUST | checker looks for heap memory allocated with malloc that hasn't been initialized before it's used |
| LDRA tools suite | 9.8.1 | 108d, 109d | Partially implemented |
| Polyspace bug finder | R2023b | CertC EXP54 | Checks for library loaded from externally controlled path and commands executed externally |
| Parasoft C/C++ test | 2023.1 | CertC EXP54 | Protect against SQL injection, file name injection, and command injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM52-CPP | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #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** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    **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;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP MEM52 | Check the return value of new. Do not allocate resources in function argument |
| PVS studio | 7.22 | V522  https://pvs-studio.com/en/blog/posts/cpp/0902/ | Partial check, might be dereferencing of null pointer |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL55-CPP | Avoid information leakage when passing a class object across a trust boundary |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, arg is value-initialized through direct initialization. Because test does not have a user-provided default constructor, the value-initialization is preceded by a zero-initialization that guarantees the padding bits are initialized to 0 before any further initialization occurs. It is akin to using std::memset() to initialize all of the bits in the object to 0. |
| #include <cstddef>    **struct** test {  **int** a;  **char** b;  **int** c;  };    // Safely copy bytes to user space  **extern** **int** copy\_to\_user(**void** \*dest, **void** \*src, std::**size\_t** size);    **void** do\_stuff(**void** \*usr\_buf) {  test arg{};    arg.a = 1;  arg.b = 2;  arg.c = 3;    copy\_to\_user(usr\_buf, &arg, **sizeof**(arg));  } |

| **Compliant Code** |
| --- |
| Padding bits can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the implementation and target memory architecture. The following solution is specific to the x86-32 architecture. |
| #include <cstddef>    **struct** test {  **int** a;  **char** b;  **char** padding\_1, padding\_2, padding\_3;  **int** c;    test(**int** a, **char** b, **int** c) : a(a), b(b),  padding\_1(0), padding\_2(0), padding\_3(0),  c(c) {}  };  // Ensure c is the next byte after the last padding byte.  static\_assert(offsetof(test, c) == offsetof(test, padding\_3) + 1,  "Object contains intermediate padding");  // Ensure there is no trailing padding.  static\_assert(**sizeof**(test) == offsetof(test, c) + **sizeof**(**int**),  "Object contains trailing padding");        // Safely copy bytes to user space.  **extern** **int** copy\_to\_user(**void** \*dest, **void** \*src, std::**size\_t** size);    **void** do\_stuff(**void** \*usr\_buf) {  test arg{1, 2, 3};  copy\_to\_user(usr\_buf, &arg, **sizeof**(arg));  } |

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

| 1. **Principles(s):** Use Effective Quality Assurance Techniques and adhere to the principle of lease priveledge |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Synopsys | 2023.9.0 | Coverity Support for SEI CERT C, C++, and Java Coding Standards | https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast/coverity-sei-cert.html# |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP | Handle all exceptions |

| **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):** Heed compiler warnings- A lot of times exceptions pertain to the compiler, catching exceptions allows compiler to decompress the stack safely |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.8.1 | 527s | Partially implemented |
| Polyspace bug finder | R2023b | CERT\_CPP ERR551 | Checks for exceptions that went unhandled |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP ERR551 | Catch exceptions, and each exception thrown has a handler of compatible type |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object oriented programming | OOP53-CPP | Write constructor member initializers in the canonical order |

| **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 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):** Keep it simple- having the declaration order of class member variables setup so that the dependency can be ordered properly in the constructor's member initializer list is a perfect example of following simple rules to better one's code |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2023.3 | CERT.OOP.CTOR.INIT\_ORDER | Partially checked |
| Astree | 22.10 | Initializer list order | Fully checked |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP OOP53 | List members in order they are declared |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input output | FIO51-CPP | Close files when they are no longer needed |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {  std::fstream file(fileName);  **if** (!file.is\_open()) {  // Handle error  **return**;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {  std::fstream file(fileName);  **if** (!file.is\_open()) {  // Handle error  **return**;  }  // ...  file.close();  **if** (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):** Practice defense in depth- application security is one layer of defense in depth, ensuring files are closed and resources are not left in use is good practice to adhere by |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klokwork | 10.4.1 | RH.LEAK | Resource leak partially checked |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2023a | CERT C++: FIO51-CPP | Checks for resource leak |
| CodeSonar | 7.0p0 | ALLOC.LEAK | leak |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | CON50-CPP | Do not destroy a mutex while it is locked |

| **Noncompliant Code** |
| --- |
| This noncompliant code example creates several threads that each invoke the do\_work() function, passing a unique number as an ID.  Unfortunately, this code contains a race condition, allowing the mutex to be destroyed while it is still owned, because start\_threads() may invoke the mutex's destructor before all of the threads have exited. |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {  std::lock\_guard<std::mutex> lk(\*pm);    // Access data protected by the lock.  }    **void** start\_threads() {  std::**thread** threads[maxThreads];  std::mutex m;    **for** (**size\_t** i = 0; i < maxThreads; ++i) {  threads[i] = std::**thread**(do\_work, i, &m);  }  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates the race condition by extending the lifetime of the mutex. |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {  std::lock\_guard<std::mutex> lk(\*pm);    // Access data protected by the lock.  }    std::mutex m;    **void** start\_threads() {  std::**thread** threads[maxThreads];    **for** (**size\_t** i = 0; i < maxThreads; ++i) {  threads[i] = std::**thread**(do\_work, i, &m);  }  } |

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

| **Principles(s):** Default deny and sanitize data- The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.  https://en.cppreference.com/w/cpp/thread/mutex |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 10.4.1 | CERT.CONC.MUTEX.DESTROY\_WHILE\_LOCKED | Do not destroy a mutex while it is locked. |
| CodeSonar | [Insert text.] | CONCURRENCY.LOCALARG | Partially checked |
| Parasoft C/C++ test | [Insert text.] | CERT\_CPP-CON50-a, CON50-CPP | Do not destroy a mutex while it is locked |

### 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 diagram above shows the dev sec ops process through a full development cycle. In pre-production and production stages there are important procedures to follow in order to set one up for a successful launch. In order to automate enforcement of the standards, clear procedures need to be put in place that developers can easily reference and follow. Tools and systems also have to be purchased and installed, and senior members need to be able to train junior employees in using them.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| DCL60-CPP | High | Probable | Medium | P12 | L1 |
| EXP53-CPP | High | Probable | Medium | P12 | L1 |
| STR50-CPP | Medium | Likely | Medium | P6 | L1 |
| EXP54-CPP | High | Probable | High | P6 | L2 |
| MEM52-CPP | High | Likely | Medium | P18 | L1 |
| DCL55-CPP | Low | Unlikely | High | P1 | L3 |
| ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| OOP53-CPP | Medium | Unlikely | Medium | P4 | L3 |
| FIO51-CPP | Medium | Unlikely | Medium | P4 | L3 |
| CON50-CPP | Medium | Probable | High | P4 | L3 |

### 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 | Data that is not actively traveling between devices or networks. Encyrption can be implemented at the source where data is generated and stored at the origin. |
| Encryption at flight | Data that is traveling from one point to another. Data might be stored in an unencrypted form at the source and destination storage, which would require encryption and decryption during transit. |
| Encryption in use | Data that is actively being accessed and processed by users. Data is most vulnerable while in this stage as it is availible and exposed to attacks. Authentication and access restrictions help enccyrption at this stage. |

https://jatheon.com/blog/data-at-rest-data-in-motion-data-in-use/

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is the process in which users are identified, it determines access to systems and relies on credentials being confirmed. Authentication can be placed at various levels. |
| Authorization | After authentication is successful, authorization can be used to determine what resources a user is allowed to access and what operations they can perform. |
| Accounting | The ability to monitor and capture events the user performs while accessing the system provides valuable information to aid in security and providing culpability. |

https://www.geeksforgeeks.org/computer-network-aaa-authentication-authorization-and-accounting/#

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 10/15/2023 | Second Revision | Jason Yaeger |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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