**oGreen Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc110763499)

[Purpose 2](#_Toc110763500)

[Scope 2](#_Toc110763501)

[Module Three Milestone 2](#_Toc110763502)

[Ten Core Security Principles 2](#_Toc110763503)

[C/C++ Ten Coding Standards 3](#_Toc110763504)

[Coding Standard 1 4](#_Toc110763505)

[Coding Standard 2 6](#_Toc110763506)

[Coding Standard 3 8](#_Toc110763507)

[Coding Standard 4 10](#_Toc110763508)

[Coding Standard 5 12](#_Toc110763509)

[Coding Standard 6 15](#_Toc110763510)

[Coding Standard 7 17](#_Toc110763511)

[Coding Standard 8 19](#_Toc110763512)

[Coding Standard 9 21](#_Toc110763513)

[Coding Standard 10 23](#_Toc110763514)

[Defense-in-Depth Illustration 25](#_Toc110763515)

[Project One 25](#_Toc110763516)

[Revise the C/C++ Standards 25](#_Toc110763517)

[Risk Assessment 25](#_Toc110763518)

[Automated Detection 25](#_Toc110763519)

[Automation 25](#_Toc110763520)

[Summary of Risk Assessments 26](#_Toc110763521)

[Create Policies for Encryption and Triple A 27](#_Toc110763522)

[Map the Principles 28](#_Toc110763523)

[Audit Controls and Management 30](#_Toc110763524)

[Enforcement 30](#_Toc110763525)

[Exceptions Process 30](#_Toc110763526)

[Distribution 31](#_Toc110763527)

[Policy Change Control 31](#_Toc110763528)

[Policy Version History 31](#_Toc110763529)

[Appendix A Lookups 31](#_Toc110763530)

[Approved C/C++ Language Acronyms 31](#_Toc110763531)

## 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 | All user input should be checked for sufficient integrity, accuracy, and structure. While untrusted sources are important to check against, being proactive in validation against all user input is a higher priority as a level of untrust is often discovered reactively. Data concerns include but are not limited to form fields, environment variables, command line arguments, login credentials, and interfaces—especially those of networks. |
| 1. Heed Compiler Warnings | Warnings should be elevated to the highest level for compile time. Warnings should not be ignored, and in the rare case where warnings need to be ignored, comments should be made at the site of warning issued and reasoning should be documented. In other words, fix the warning by fixing the code using static and/or dynamic analyses tools rather than silencing the warning, if possible. |
| 1. Architect and Design for Security Policies | Design tools in architecture of software such as flow charts, pseudocode, mind maps, sequence graphs, UML charts, data models and the like should all take proper security policies into sincere consideration such that (i.e., validation, assertion, authentication, etc. are included in the design(s). |
| 1. Keep It Simple | Keep logic and design simple. Avoid obfuscation. Code each function for a single task if possible. Comment code and provide reference for code that is not yours to help others decipher any issues with it later. Organize functions, classes, data structures, and directories to optimize readability and maintainability. Avoid specificity in variable names within reason to optimize reusability of code which simplifies overall development. |
| 1. Default Deny | Ensure that access is a result of permissible action rather than exclusion of non-permissible action. Take a ‘guilty until proven innocent’ approach where checking for innocence to gain access is of highest priority. In other words, by default, deny all except the undeniable. |
| 1. Adhere to the Principle of Least Privilege | Only provide the least set of privileges required to complete the job. Elevated and/or administrative privileges should only be available for ample time to complete the task at hand. The purpose is to minimize damages, typically using arbitrary code, caused by a ‘mole’ or ‘insider’ with elevated privileges or someone who might social engineer, tailgate, or shoulder-surf someone with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | Clean the data of unexpected characters, artifacts, commands, arguments, or other malicious arbitrary code before allowing the data to pass through other systems or networks. Validating input is one of the first lines of defense here but it is equally important to maintain that validation to provide defense-in-depth thereby accounting for unknown loopholes and uncaught exceptions in the systems overall. |
| 1. Practice Defense in Depth | Create, implement, practice, and maintain multiple layers of security such that when one layer fails there are several more layers to stop the penetration. An example would be allowing a user to register, implementing email verification, forcing a user to input a secure password with specific rules of creating the password, implementing two-factor authentication for user logins, implementing a (i.e.) reCAPTCHA to verify the user is not a robot, and questioning a user login from an unfamiliar IP address/location. Then, on top of all that, deny any input that is not the username and password (i.e., implement prevention of SQL injection and other known vulnerabilities). There is no single solution to eliminate all threats, so if there is a doubt, design a layer(s) of defense to accommodate. |
| 1. Use Effective Quality Assurance Techniques | Use several types of testing such as functional, non-functional, compatibility, usability, performance, integration, vulnerability, acceptance, system, and unit testing where appropriate to minimize vulnerabilities and optimize product quality. Utilize peer-to-peer and technical code reviews, audits and inspections, walkthroughs, and supervisorial reviews in keeping systems and code secure. Sustain objectivity during all reviews. |
| 1. Adopt a Secure Coding Standard | Do not engage in development using any language or platform without adopting a relevant and secure coding standard for that language and/or platform. This is important because it is highly likely that when using various systems, especially when all are connected, that different languages will be present within those systems. The absence of secure coding standards can be detrimental to a defense-in-depth policy because of the holes and back doors it can create in a system. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Never qualify a reference type with const or volatile. |

| **Noncompliant Code** |
| --- |
| A const-qualified reference to a char is formed instead of a reference to a const-qualified char resulting in undefined behavior. In Microsoft Visual Studio 15, this results in a warning diagnostic, and in Clang 3.9, this results in a fatal diagnostic—both messages imply that qualifiers should not be applied to references. |
| #include <iostream>    **void** f(**char** c) {  **char** &**const** p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This example removes the const qualifier resulting in a well-formed solution. |
| #include <iostream>    **void** f(**char** c) {  **char** &p = c;    p = 'p';    std::cout << c << std::endl;  } |

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

| **Principles(s):** 2. Heed Compiler Warnings, 7. Sanitize Data Sent to Other Systems  Undefined behavior can cause unintentional values to be stored thereby causing possible loss of data integrity. This is one of many reasons why sanitizing data is important because of these unexpected values. Since noncompliance here results in a warning diagnostic, it is important to heed the compiler warning and make the adjustment to compliant code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 | Never qualify a reference type with const or volatile. |
| Klocwork | 2022.2 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE | Never qualify a reference type with ‘const’ or ‘volatile’ |
| Polyspace Bug Finder | R2022a | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| Clang | 3.9 | Clang static analyzer | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not attempt to modify string literals. **Note**: it is noted in the SEI CERT CPP Coding Standard that this standard, albeit a SEI CERT C Coding Standard, still applies to C++. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the char pointer str is initialized to the address of “string literal.” The following attempt to modify the string literal results in undefined behavior. |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Instead, as a compliant solution, the array of chars should be initialized with values thereby defining the size of the array. This example creates a copy of the string literal in the memory allocated to str which allows string stored in str to be modified safely. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** 4. Keep it simple., 8. Practice Defense in Depth  Modifying string literals can lead to abnormal program terminations and possibly denial-of-service attacks. It is important to initialize data in a simple, proper manner to avoid undefined behaviors and to make the code readable and maintainable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | String-literal-modification | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-STR30-a  CERT\_C-STR30-b | A string literal shall not be modified.  Do not modify string literals. |
| Polyspace Bug Finder | R2202a | CERT C: Rule STR30-C | Checks for writing to const qualified object (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Use valid references, pointers, and iterators to reference elements of a basic\_string. |

| **Noncompliant Code** |
| --- |
| This noncompliant example intends to copy input to a string and replace any semicolon characters with spaces. However, the iterator loc is invalid at the first call to insert(). Thus, the calls to insert() thereafter are undefined. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a simple, standard, generic algorithm to perform the replacement with std::replace(). |
| #include <algorithm>  #include <string>    **void** f(**const** std::string &input) {    std::string email{input};    std::replace(email.begin(), email.end(), ';', ' ');  } |

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

| **Principles(s):** 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques  Using an invalid reference, pointer, or iterator to a string object could allow an attacker to run arbitrary code. It is important to run exhaustive testing to validate these references, pointers and/or iterators. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.0p0 | ALLOC.UAF | Use After Free |
| Coverity | 2021.12.0 | STR52-CPP | Use valid references, pointers, and iterators to reference elements of a basic\_string. |
| Helix QAC | 2022.2 | C++4746, C++4747, C++4748, C++4749 | Use valid references, pointers, and iterators to reference elements of a basic\_string. |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Exclude user input from format strings. Note: this is another SEI C Coding Standard that applies to C++. |

| **Noncompliant Code** |
| --- |
| This noncompliant example contains a function incorrect\_password() that is called during identification and authentication of a user, and the function returns an error message to the user if the credentials are not found. The problem here is the addition operations are not checked for integer overflow because of an expected format of 256 chars or less. More importantly, the potentially untrusted user input is passed to snprintf() which is commonplace to be used for conveying messages to multiple locations in the system. Further, the untrusted data is thus evaluated unnecessarily. |
| #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** |
| --- |
| The compliant solution does not evaluate the untrusted data and, instead, sends the error message with the untrusted data right to the output in a message to the user regarding its failure. |
| #include <stdio.h>    **void** incorrect\_password(**const** **char** \*user) {  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **fprintf**(stderr, msg\_format, user);  } |

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

| **Principles(s):** 4. Keep it Simple, 5. Default Deny, 6. Adhere to the Principle of Least Privilege, 8. Practice Defense in Depth  Failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process. This attack could be especially dangerous if the user in question had unnecessary administrative privileges by default which is why it is important to define and implement user roles throughout the system.  Instead of validating user input, we want to avoid the possibility of malicious input. Simplicity here is key in relaying the message as a standard error to the user. Further, notice we are denying the user input by default because validation of this input is unnecessary to relay the message. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | Astrée Static Analyzer | Supported via stubbing/taint analysis |
| Code Sonar | 7.0p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Parasoft C/C++ test | 2022.1 | 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/wprintf with only two parameters, when second parameter is a variable.  Never use unfiltered data from an untrusted user as the format parameter. |
| Polyspace Bug Finder | R2202a | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Detect and handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the same expression g() performs two memory allocations for A and B, respectively. An exception thrown as a result of the calls to either A or B could result in a memory leak. |
| **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(A \*, B \*);  **void** f() {    g(**new** A, **new** B);  } |

| **Compliant Code** |
| --- |
| This compliant solution removes memory allocation altogether and instead passes the objects by reference. |
| **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(A &a, B &b);  **void** f() {    A a;    B b;    g(a, b);  } |

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

| **Principles(s):** 2. Heed Compiler Warnings, 4. Keep it Simple, 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques  Failing to detect allocation failures can lead to abnormal program termination and denial-of-service attacks. If the vulnerable program references memory offset from the return value, an attacker can exploit the program to read or write arbitrary memory. This vulnerability has been used to execute arbitrary code [VU#159523]. It is important to not only detect the allocation failures, but to also heed the compiler warning considering these failures. As we see in the compliant code, the code is kept simple by removing allocation altogether. Effective quality assurance techniques may be used to assert if any pointers returned by malloc() are NULL or not. |
| --- |

**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. |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new.  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined. |
| Parasoft Insure++ | 7.5.4 | LEAK\_FREE  LEAK\_RETURN  LEAK\_SCOPE | Runtime Detection  Occurs when you free a block of memory that contains pointers to other memory blocks. If there are no other pointers that point to these secondary blocks then they are permanently lost and will be reported by Insure++.  Occurs when a function returns a pointer to an allocated block of memory, but the returned value is ignored in the calling routine.  Occurs when a function contains a local variable that points to a block of memory, but the function returns without saving the pointer in a global variable or passing it back to its caller. |
| Polyspace Bug Finder | R2022a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use a static assertion to test the value of a constant expression. |

| **Noncompliant Code** |
| --- |
| This noncompliant example attempts to use the assert() macro to validate the property of a memory-mapped struct named ‘timer.’ However, if the function func() is never called, the assertion is never executed thus occurring at runtime only. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| The compliant and portable solution would be to allow diagnostics to take place at compile time by using a static assertion resulting in proactive debugging versus a malfunction that may not be caught before it is too late. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    static\_assert(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**),                "Structure must not have any padding"); |

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

| **Principles(s):** 2. Heed Compiler Warnings, 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques  Static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities at compile time. The absence of static assertions, however, does not mean that code is incorrect. It is important to watch for these vulnerabilities at compile time and take action to make the code compliant with assertions and exhaustive testing. It is important to remember that failing to validate expected values often leads to vulnerabilities, thus doing so is considered a layer of defense. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-006-CPP | Do not leak resources when handling exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, when process\_item() throws an exception, pst fails to be released properly. This results in a resource leak. The consensus is for any constructors or objects left in an incomplete state to throw exceptions rather than exit normally and remain in the incomplete state. |
| #include <new>    **struct** SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.  **void** process\_item() noexcept(**false**);  };    **void** f() {    SomeType \*pst = **new** (std::**nothrow**) SomeType();  **if** (!pst) {      // Handle error  **return**;    }    **try** {      pst->process\_item();    } **catch** (...) {      // Process error, but do not recover from it; rethrow.  **throw**;    }  **delete** pst;  } |

| **Compliant Code** |
| --- |
| The ideal compliant solution is to utilize the *Resource Acquisition Is Initialization* (RAII) technique which ‘binds the life cycle of a resource acquired before use to the lifetime of an object’: <https://en.cppreference.com/w/cpp/language/raii>. The compliant solution below uses RAII to establish a destructor alongside the objects constructor which is then called whether an exception is thrown or not. |
| **struct** SomeType {    SomeType() noexcept; // Performs nontrivial initialization.    ~SomeType(); // Performs nontrivial finalization.    **void** process\_item() noexcept(**false**);  };    **void** f() {    SomeType st;  **try** {      st.process\_item();    } **catch** (...) {      // Process error, but do not recover from it; rethrow.  **throw**;    } // After re-throwing the exception, the destructor is run for st.  } // If f() exits without throwing an exception, the destructor is run for st. |

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

| **Principles(s):** 3. Architect and Design for Security Policies, 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques, 10. Adopt a Secure Coding Standard  Memory and other resource leaks will eventually cause a program to crash. If an attacker can provoke repeated resource leaks by forcing an exception to be thrown through the submission of suitably crafted data, then the attacker can mount a denial-of-service attack. It is important throughout architecture, design, and coding standards to be proactive in ensuring destructors and memory deallocations are properly implemented using QA techniques, exhaustive testing, and otherwise adherent to code compliance. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | ALLOC.LEAK | Leak |
| LDRA tool suite | 9.7.1 | 50 D | Partially Implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-ERR57-a | Ensure resources are freed |
| Polyspace Bug Finder | R2022a | CERT C++: ERR57-CPP | * Resource leak caused by exception * Object left in partially initialized state * Bad allocation in constructor |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exception Handling | STD-008-CPP | Handle all exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, no exceptions are caught throughout the program. Since there does not exist a matching handler for any exception thrown by f(), std::terminate() is called and the program ends abruptly. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, main() handles all exception with catch(...) allowing the stack to effectively manage external resources and enables the program to end normally. |
| **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):** 3. Architect and Design for Security Policies, 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques, 10. Adopt a Secure Coding Standard  Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. This coding standard works in conjunction with the principles of Coding Standard 7. It is important to implement exception handling from design and architecture all the way through deployment and maintenance. Notably, all errors that can be handled directly should take precedence before any catch-all (i.e., “catch(...){}”) is used. Handling all exceptions is a vital layer of defense that allows developers to be proactive in error-handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | main-function-catch-all  early-catch-all | Partially checked |
| LDRA tool suite | 9.71 | 527 S | Partially implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions.  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point. |
| Polyspace Bug Finder | R2022a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Enumeration | STD-009-CPP | Do not cast to an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| This noncompliant example attempts to evaluate the range of acceptable enumeration values AFTER casting to the enumeration type, which may not be able to process the given integer value. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| This compliant solution evaluates the range of acceptable enumeration values BEFORE performing the conversion thereby preventing any unspecified value as a result of the conversion. This solution succeeds using bounds checking and explicit validation versus other compliant solutions that allow any value from the parameter to be converted. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** 1. Validate input data, 7. Sanitize Data Sent to Other Systems, 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques    It is possible for unspecified values to result in a buffer overflow, leading to the execution of arbitrary code by an attacker. However, because enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will result in data integrity violations rather than arbitrary code execution.  It is important to validate acceptable enumeration values, especially if those values are the result of user input. Exhaustive testing should be performed in any case to ensure any arbitrary code does not leak into other systems. Assertions and other QA techniques should be utilized in validations and testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | Do not cast to an out-of-range enumeration value. |
| CodeSonar | 7.0p0 | LANG.CAST.COERCE  LAN.CAST.VALUE | Coercion Alters Value.  Cast Alters Value. |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PVS-Studio | 7.19 | V1016 | Do not cast to an out-of-range enumeration value. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | STD-010-CPP | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an fstream object is constructed and used to open the file buffer. When function f returns, the standard terminate() does not call a destructor, so the file buffer 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, the file stream is closed using close() before terminate() is called making sure that file resources are closed properly. |
| #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):** 2. Heed Compiler Warnings, 8. Practice Defense in Depth, 9. Use Effective Quality Assurance Techniques  Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination. Compilers will usually let you know if an object was not properly closed, so it is important to pay attention and take action to correct the issue considering these warnings.  Utilize the *Resource Acquisition Is Initialization* (RAII) technique which ‘binds the life cycle of a resource acquired before use to the lifetime of an object’: <https://en.cppreference.com/w/cpp/language/raii>. RAII is an effective quality assurance technique. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2022.1 | CERT\_C-FIO42-a | Ensure resources are freed. |
| PC-lint Plus | 1.4 | 429 | Partially Supported |
| Polyspace Bug Finder | R2022a | CERT C: Rule FIO42-C | Checks for resource leak (rule partially covered) |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Given the diagram above, this policy defines standards that aim to modify the existing DevOps process in converting to a DevSecOps policy standard. Namely, the standards focus on implementations in the following *Pre-production* sections: *Assess and Plan*, *Design*, *Build*, *Verify and test*. However, it is also important to be aware of the changes when executing the remainder of the DevSecOps policy in the *Production* sections: *Transition and health check*, *Monitor and detect*, *Respond*, and *Maintain and stabilize*. In other words, this policy creates a standard of being proactive in architecture, design, building, and testing the application before reaching the production cycle. The standards should then be continued to be maintained and adjusted accordingly in the case of any malicious behavior, attacks, penetration test results, or updated standards.

Each standard in this policy includes corresponding principles—defined at the beginning of this policy. These principles define what stage the standard should be implemented, and the compliant code solutions define how. You will notice that two principles (8 and 9) correspond with a majority of standards in this policy. This is intentional to drive the idea home that practicing Defense in Depth and implementing effective Quality Assurance techniques are imperative in attaining an exceptional DevSecOps policy standard at the very least.

Notably, this becomes extremely significant when implementing Lightweight Directory Active Protocol (LDAP) and Active Directory (AD) services as LDAP queries string-based objects from the AD which often contains sensitive information. Principle of Least Privilege takes precedence here in this implementation as well because of the nature of the data. The other standards listed are no less important, but the overarching principles of layered defense and exhaustive testing should be noted.

### 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 | Low | Unlikely | Low | P3 | L3 |
| STD-002-CPP | Low | Likely | Low | P9 | L2 |
| STD-003-CPP | High | Probable | High | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | High | P2 | L3 |
| STD-008-CPP | Low | Probable | Medium | P4 | L3 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Medium | Unlikely | Medium | 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 | Encryption at rest is defined as protecting the data ‘on disk’ or in its otherwise stationary form (i.e., cloud-based application would not be ‘on disk’). The purpose of encrypting data in rest is to add another layer of defense (i.e., defense-in-depth) in the event an attacker either physically obtains the hard drive (typically the result of someone with inside access), or someone who can penetrate defenses leading up to and including the data at rest. The policy applies because security and data breaches occur at all layers of defense, so accounting for physical and/or raw data access is a vital implementation. |
| Encryption at flight | Encryption at flight is defined as protecting the data in transit from point-to-point on any network and/or internet connection. The purpose of encrypting data at flight is to add another layer of defense (i.e., defense-in-depth) in the event an attacker is attempting to intercept and view and/or steal sensitive data, spoof data with their own defined headers in browser requests to/from databases, or gain access to administrative privileges. This policy applies because modern applications are usually sending data in some shape or form either over the internet or across an internal/external network of some sort, and if data can be transmitted, it can also be intercepted and/or modified in transit. Additionally, this is one of many reasons why *Sanitizing Data Sent to Other Systems* is an important principle. |
| Encryption in use | Encryption in use is defined as protecting the data within random-access memory (RAM), CPU caches, or CPU registers or otherwise any data being currently processed by applications in a cloud service. The purpose of encrypting data in use is to add another layer of defense (i.e., defense-in-depth) in the event an application becomes victim to any rootkits, boot kits, cold boot attacks, or other malicious hardware attacks. For example, someone with access to RAM could parse the memory to obtain the key to decrypt the ‘encryption in rest’ data. This policy applies because data in use may contain many different forms of sensitive data including but not limited to business intelligence, personal information, keys for encryption, and digital certificates. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The role of authentication is to evaluate the type of user in ensuring the user is a human as desired—namely, a user who is registered and interested in using the website as a consumer. A best practice often involves eliminating guest checkouts and implementing external (i.e., two-factor) authentications to further verify account creation and/or changes to private user data. |
| Authorization | The role of authorization is to determine whether a user on the website has the proper permission(s) to conduct any given behavior within the site. In other words, if a website eliminates admin access from registered users but forgot to eliminate admin access from guests, any random user who did not login (including bots) could be granted admin access. Best practice is the Principle of Least Privilege which means a developer should never grant more access to a user than is needed to complete the intended behavior of that user role. |
| Accounting | The role of accounting is accountability. It is important to implement terms of service and conditions to the user before a user engages in activity on a website to be upfront about what behaviors are allowed, what a user’s obligations and responsibilities are defined as, and to be clear about actions that may take place as a result of good and/or bad user behavior. For example, terms and conditions should include how the website will use a user’s personal information as well as what the consequences are in the event a user is held responsible for a security, data, and/or policy breach. Best practice is to at the very least ensure a user scrolls to the bottom of the terms and conditions before an “I Agree” button becomes available to press. |

**\***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 | 07/14/2022 | * Defined Principles. * Defined Coding Standards. * Defined both Noncompliant and Compliant Code solutions for each standard. | Mark Meyer | Professor Ahlam Alhweiti, M.S |
| 1.2 | 08/07/2022 | * Completed Principle(s) associations, Threat Levels and Automation tools within each Coding Standard. * Completed Summary of Risk Assessments. * Completed guidance on converting DevOps to DevSecOps policy. * Completed Encryption and Triple A policies. | Mark Meyer | Professor Ahlam Alhweiti, M. S |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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