**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. |
| --- | --- |
| Validate Input Data | Data from untrusted sources need their input validated at all times. Validation must happen from external data sources, command line arguments, environmental files, and user-controlled files. Being suspicious from the start of these inputs is crucial in maintaining robust security. |
| Heed Compiler Warnings | Developers must set compilers to the highest warning levels to eliminate all possible warnings in their code. Aligning this sensitivity to the max will allow code to be free from any potential vulnerabilities. Using static and dynamic testing tools will aid in eliminating further security flaws. |
| Architect and Design for Security Policies | A system's architecture should be built with security in mind. Breaking the system into smaller interconnected sections with separate privilege sets will ensure enforced security policies. |
| Keep It Simple | Complex system designs lead to an increased likelihood of errors. Ensuring a simple architecture and a secure environment from implementation, configuration, and use. Simplified security mechanisms lead to less complex work while achieving the required level of assurance. |
| Default Deny | Access must be defaulted to deny. Protection systems should have approval based on set permissions instead of exclusions. |
| Adhere to the Principle of Least Privilege | All requested access must happen at the lowest possible level needed for the role. Instead of granting all permissions for requests, only what is necessary to accomplish the task is required—reducing opportunities for attackers to take advantage of elevated permissions. |
| Sanitize Data Sent to Other Systems | Any data passed to other connected subsystems must be sanitized before sending. This ensures an attacker can not invoke unused functionality in these components to perform an injection attack. |
| Practice Defense in Depth | Developers must implement multiple layers of defenses to prevent security flaws from exploitation while reducing vulnerability repercussions. Managing risk will provide a secure environment. |
| Use Effective Quality Assurance Techniques | Developers should ensure proper testing for quality assurance reasons. Implementing fuzz testing, penetration testing, and source code audits are effective strategies to fortify code from vulnerabilities. |
| Adopt a Secure Coding Standard | Organizations should establish secure coding standards for their development environment and coding practices to ensure consistent security across their entire codebase and application. |

### 

### 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** | **Do not write syntactically ambiguous declarations** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not write a syntactically ambiguous declaration. With the advent of uniform initialization syntax using a braced-init-list, there is now syntax that unambiguously specifies a declaration instead of an expression statement. Declarations can also be disambiguated by using nonfunction-style casts, by initializing using =, or by removing extraneous parenthesis around the parameter name. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an anonymous local variable of type std::unique\_lock is expected to lock and unlock the mutex m by virtue of RAII. However, the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named m and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex>(m);  shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lock object is given an identifier (other than m) and the proper converting constructor is called. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex> lock(m);  shared\_resource += 42;  } |

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

| **Principles(s):** 4. Keep it Simple and 2. Heed Complier Warnings.  The keep it simple principle will promote a clear and straightforward solution to the code that is easy to maintain and secure throughout its life. Heeding compiler warnings will help with avoiding ambiguous declarations by reducing compiler warnings and errors and potentially stopping a vulnerability from surfacing. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |
| LDRA tool suite | 9.7.1 | **296 S** | Partially implemented |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-DCL53-a** **CERT\_CPP-DCL53-b** **CERT\_CPP-DCL53-c** | Parameter names in function declarations should not be enclosed in parentheses. Local variable names in variable declarations should not be enclosed in parentheses. Avoid function declarations that are syntactically ambiguous. |
| Polyspace Bug Finder | R2023a | [CERT C++: DCL53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl53cpp.html) | Checks for declarations that can be confused between:   * Function and object declaration * Unnamed object or function parameter declaration   Rule fully covered. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use correct integer precisions** |
| --- | --- | --- |
| **Data Value** | [STD-002-C] | Integer types in C have both a size and a precision. The size indicates the number of bytes used by an object and can be retrieved for any object or type using the sizeof operator. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits.  Padding bits contribute to the integer's size, but not to its precision. Consequently, inferring the precision of an integer type from its size may result in too large a value, which can then lead to incorrect assumptions about the numeric range of these types. Programmers should use correct integer precisions in their code, and in particular, should not use the sizeof operator to compute the precision of an integer type on architectures that use padding bits or in strictly conforming (that is, portable) programs. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example illustrates a function that produces 2 raised to the power of the function argument. To prevent undefined behavior in compliance with INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand, the function ensures that the argument is less than the number of bits used to store a value of type unsigned int. |
| #include <limits.h>    unsigned int pow2(unsigned int exp) {  if (exp >= sizeof(unsigned int) \* CHAR\_BIT) {  /\* Handle error \*/  }  return 1 << exp;  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a popcount() function, which counts the number of bits set on any unsigned integer, allowing this code to determine the precision of any integer type, signed or unsigned. |
| |  | | --- | | #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  **size\_t** popcount(uintmax\_t num) {  **size\_t** precision = 0;  **while** (num != 0) {  **if** (num % 2 == 1) {        precision++;      }      num >>= 1;    }  **return** precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) | |

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

| **Principles(s):** 1. Validate Input Data  By using the correct integer precisions we can make sure data is being represented as accurately as possible. This will assist in minimizing the risks of data being misinterpreted and prevent vulnerabilities from occurring. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **P2** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **LANG.ARITH.BIGSHIFT** | Shift Amount Exceeds Bit Width |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.3 | **C0582**  **C++3115** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-INT35-a** | Use correct integer precisions when checking the right hand operand of the shift operator |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule INT35-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint35c.html) | Checks for situations when integer precisions are exceeded (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-C] | At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See undefined behavior 33.) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior: |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can 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):** 2. Heed Compiler Warnings and 4. Keep it Simple  Compiler warnings will populate when string literals are modified allowing the developer to minimize the threat. By keeping it simple, the code can be more straightforward and less prone to errors since it will not be modified. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **string-literal-modfication** **write-to-string-literal** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR30** | Fully implemented |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **489, 1776** | Partially supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule STR30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr30c.html) | Checks for writing to const qualified object (rule fully covered) |

#### 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 . 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]. 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. |

| **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 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);  } |

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

| **Principles(s):** 3. Architect and Design for Security Policies and 8. Practice Defense in Depth  Ensuring format strings are protected from malicious values in the design of the software or system will inherently protect the string from being misused. By excluding user input from format strings will be practicing defense in depth through the creation of another layer of protection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-FIO30** | Partially implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **IO.INJ.FMT** **MISC.FMT** | Format string injection Format string |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **86 D** | Partially Implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule FIO30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulefio30c.html) | Checks for tainted string format (rule partially covered) |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Only free memory allocated dynamically** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C] | Freeing memory that is not allocated dynamically can result in heap corruption and other serious errors. Do not call free() on a pointer other than one returned by a standard memory allocation function, such as malloc(), calloc(), realloc(), or aligned\_alloc().  A similar situation arises when realloc() is supplied a pointer to non-dynamically allocated memory. The realloc() function is used to resize a block of dynamic memory. If realloc() is supplied a pointer to memory not allocated by a standard memory allocation function, the behavior is undefined. One consequence is that the program may terminate abnormally.  This rule does not apply to null pointers. The C Standard guarantees that if free() is passed a null pointer, no action occurs |

| **Noncompliant Code** |
| --- |
| This noncompliant code example sets c\_str to reference either dynamically allocated memory or a statically allocated string literal depending on the value of argc. In either case, c\_str is passed as an argument to free(). If anything other than dynamically allocated memory is referenced by c\_str, the call to free(c\_str) is erroneous. |
| #include <stdlib.h>  #include <string.h>  #include <stdio.h>    enum { MAX\_ALLOCATION = 1000 };    int main(int argc, const char \*argv[]) {  char \*c\_str = NULL;  size\_t len;    if (argc == 2) {  len = strlen(argv[1]) + 1;  if (len > MAX\_ALLOCATION) {  /\* Handle error \*/  }  c\_str = (char \*)malloc(len);  if (c\_str == NULL) {  /\* Handle error \*/  }  strcpy(c\_str, argv[1]);  } else {  c\_str = "usage: $>a.exe [string]";  printf("%s\n", c\_str);  }  free(c\_str);  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates the possibility of c\_str referencing memory that is not allocated dynamically when passed to free(): |
| #include <stdlib.h>  #include <string.h>  #include <stdio.h>    enum { MAX\_ALLOCATION = 1000 };    int main(int argc, const char \*argv[]) {  char \*c\_str = NULL;  size\_t len;    if (argc == 2) {  len = strlen(argv[1]) + 1;  if (len > MAX\_ALLOCATION) {  /\* Handle error \*/  }  c\_str = (char \*)malloc(len);  if (c\_str == NULL) {  /\* Handle error \*/  }  strcpy(c\_str, argv[1]);  } else {  printf("%s\n", "usage: $>a.exe [string]");  return EXIT\_FAILURE;  }  free(c\_str);  return 0;  } |

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

| **Principles(s):** 6. Adhere to the Principle of Least Privilege  Limiting the system to specific services or functions to free memory and doing so dynamically will ensure the system operates within the minimum required access and safeguard against unwanted memory operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Tool | Version | Checker | Description |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **invalid-free** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM34** | Can detect memory deallocations for stack objects |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | **clang-analyzer-unix.Malloc** | Checked by clang-tidy; can detect some instances of this rule, but does not detect all |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see MSC11-C. Incorporate diagnostic tests using assertions). The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #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** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** 9. Use Effective Quality Assurance Techniques  Using static assertions is a worthwhile quality assurance measure. It helps ensure that only certain conditions are met before the code even runs. This fortifies the code even further across development and testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Honor exception specifications** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | If a function throws an exception other than one allowed by its *exception-specification*, it can lead to an [implementation-defined](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-implementation-definedbehavior) termination of the program ([except.spec], paragraph 9).  If a function declared with a *dynamic-exception-specification* throws an exception of a type that would not match the *exception-specification*, the function std::unexpected() is called. The behavior of this function can be overridden but, by default, causes an exception of std::bad\_exception to be thrown. Unless std::bad\_exception is listed in the *exception-specification*, the function std::terminate() will be called.  Similarly, if a function declared with a *noexcept-specification* throws an exception of a type that would cause the *noexcept-specification* to evaluate to false, the function std::terminate() will be called.  Calling std::terminate() leads to implementation-defined termination of the program. To prevent [abnormal termination](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination) of the program, any function that declares an *exception-specification* should restrict itself, as well as any functions it calls, to throwing only allowed exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) noexcept(true) {  v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) {  v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):** 4. Keep it Simple and 6. Adhere to the Principle of Least Privilege  If exception specifications are honored, developers can ensure the code is clear about what caused the exception, making maintenance and future work easier. Also, by ensuring a function restricts itself to only allowed exceptions, allows the system to operate under predetermined boundaries. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | **LANG.STRUCT.EXCP.THROW** | Use of throw |
| [Parasoft C/C++Test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR55-a** | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: ERR55-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr55cpp.html) | Checks for noexcept functions exiting with exception (rule fully covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **unhandled-throw-noexcept** | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Use valid iterator ranges** |
| --- | --- | --- |
| Iterators | [STD-008-CPP] | When iterating over elements of a container, the iterators used must iterate over a valid range. An iterator range is a pair of iterators that refer to the first and past-the-end elements of the range respectively.  A valid iterator range has all of the following characteristics:   * Both iterators refer into the same container. * The iterator representing the start of the range precedes the iterator representing the end of the range. * The iterators are not invalidated, in conformance with CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container.   An empty iterator range (where the two iterators are valid and equivalent) is considered to be valid.  Using a range of two iterators that are invalidated or do not refer into the same container results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, std::for\_each() compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results in undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the iterator values passed to std::for\_each() are passed in the proper order. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):** 6. Adhere to the Principle of Least Privilege  Iterators are prevented from doing any unauthorized operations or inducing any errors that could result in vulnerabilities by only allowing them to function inside the boundaries and ranges that correspond to them. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | **LANG.MEM.BO** | Buffer Overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-CTR53-a** **CERT\_CPP-CTR53-b** | Do not use an iterator range that isn't really a range Do not compare iterators from different containers |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C++: CTR53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr53cpp.html) | Checks for invalid iterator range (rule partially covered). |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.26 | [**V539**](https://pvs-studio.com/en/docs/warnings/v539/)**,**[**V662**](https://pvs-studio.com/en/docs/warnings/v662/)**,**[V789](https://pvs-studio.com/en/docs/warnings/v789/) |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Ensure that division and remainder operations do not result in divide-by-zero errors** |
| --- | --- | --- |
| Operation Errors | [STD-009-C] | The result of the / operator is the quotient from the division of the first arithmetic operand by the second arithmetic operand. Division operations are susceptible to divide-by-zero errors. Overflow can also occur during two's complement signed integer division when the dividend is equal to the minimum (most negative) value for the signed integer type and the divisor is equal to −1. (See INT32-C. Ensure that operations on signed integers do not result in overflow.) |

| **Noncompliant Code** |
| --- |
| This noncompliant code example prevents signed integer overflow in compliance with INT32-C. Ensure that operations on signed integers do not result in overflow but fails to prevent a divide-by-zero error during the division of the signed operands s\_a and s\_b: |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_a == LONG\_MIN) && (s\_b == -1)) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution tests the division operation to guarantee there is no possibility of divide-by-zero errors or signed overflow: |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

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

| **Principles(s):** 2. Heed Compiler Warnings  Compiler warnings can populate when there are unsafe division operations. Rectifying these warnings and errors can ensure the code will be free of vulnerabilities and unwanted behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **int-division-by-zero**  **int-modulo-by-zero** | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **LANG.ARITH.DIVZERO** **LANG.ARITH.FDIVZERO** | Division by zero Float Division By Zero |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **DIVIDE\_BY\_ZERO** | Fully implemented |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **zerodiv** **zerodivcond** | Context sensitive analysis of division by zero Not detected for division by struct member / array element / pointer data that is 0 Detected when there is unsafe division by variable before/after test if variable is zero |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Ensure size arguments for variable length arrays are in a valid range** |
| --- | --- | --- |
| Arrays | [STD-010-C] | Variable length arrays (VLAs), a conditionally supported language feature, are essentially the same as traditional C arrays except that they are declared with a size that is not a constant integer expression and can be declared only at block scope or function prototype scope and no linkage. When supported, a variable length array can be declared where the integer expression size and the declaration of vla are both evaluated at runtime. If the size argument supplied to a variable length array is not a positive integer value, the behavior is undefined. (See undefined behavior 75.) Additionally, if the magnitude of the argument is excessive, the program may behave in an unexpected way. An attacker may be able to leverage this behavior to overwrite critical program data [Griffiths 2006]. The programmer must ensure that size arguments to variable length arrays, especially those derived from untrusted data, are in a valid range.  Because variable length arrays are a conditionally supported feature of C11, their use in portable code should be guarded by testing the value of the macro \_\_STDC\_NO\_VLA\_\_. Implementations that do not support variable length arrays indicate it by setting \_\_STDC\_NO\_VLA\_\_ to the integer constant 1. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a variable length array of size size is declared. The size is declared as size\_t in compliance with INT01-C. Use rsize\_t or size\_t for all integer values representing the size of an object. |
| #include <stddef.h>    extern void do\_work(int \*array, size\_t size);    void func(size\_t size) {  int vla[size];  do\_work(vla, size);  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures the size argument used to allocate vla is in a valid range (between 1 and a programmer-defined maximum); otherwise, it uses an algorithm that relies on dynamic memory allocation. The solution also avoids unsigned integer wrapping that, given a sufficiently large value of size, would cause malloc to allocate insufficient storage for the array. |
| #include <stdint.h>  #include <stdlib.h>    enum { MAX\_ARRAY = 1024 };  extern void do\_work(int \*array, size\_t size);    void func(size\_t size) {  if (0 == size || SIZE\_MAX / sizeof(int) < size) {  /\* Handle error \*/  return;  }  if (size < MAX\_ARRAY) {  int vla[size];  do\_work(vla, size);  } else {  int \*array = (int \*)malloc(size \* sizeof(int));  if (array == NULL) {  /\* Handle error \*/  }  do\_work(array, size);  free(array);  }  } |

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

| **Principles(s):** 1. Validate Input Data and 4. Keep it Simple  Ensuring the size arguments of variable length arrays are in the valid range will prevent unwanted behavior. Keeping the code simple will also allow for the possibility of unpredictable behavior or potential security issues to be minimized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **ALLOC.SIZE.IOFLOW** **ALLOC.SIZE.MULOFLOW** **MISC.MEM.SIZE.BAD** | Integer Overflow of Allocation Size Multiplication Overflow of Allocation Size Unreasonable Size Argument |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **REVERSE\_NEGATIVE** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-ARR32-a** | Ensure the size of the variable length array is in valid range |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **9035** | Assistance provided |

### 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.

Automation inside Green Pace’s DevSecOps pipeline is critical for improving security and streamlining compliance. Beginning with the “Assess and Plan” phase, automated technologies may continuously assess errors and reprioritize work. This ensures that high-risk items receive prompt attention. To enforce code standards and search for vulnerabilities, SonarSource, and other similar tools can be included during the “Design” and “Build” phases. The “Verify and Test” steps benefit greatly from automated vulnerability scanning and security testing, whereas MDR or SIEM technologies provide real-time threat detection in the “Monitor and Detect” phase. Automation in the “Respond” phase can quickly thwart attacks and assure adherence to the security baselines during maintenance. Green Pace will improve its security posture, decrease manual errors, and respond quickly to emerging threats by incorporating these tools and feedback mechanisms into its existing infrastructure.

### 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 | P2 | L3 |
| STD-002-C | Low | Unlikely | Medium | **P2** | **L3** |
| STD-003-C | Low | Likely | Low | **P9** | **L2** |
| STD-004-C | High | Likely | Medium | **P18** | L1 |
| STD-005-C | High | Likely | Medium | **P18** | **L1** |
| STD-006-C | Low | Unlikely | High | **P1** | **L3** |
| STD-007-CPP | Low | Likely | Low | **P9** | **L2** |
| STD-008-CPP | High | Probable | High | **P6** | **L2** |
| STD-009-C | Low | Likely | Medium | **P6** | **L2** |
| STD-010-C | High | Probable | High | **P6** | **L2** |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This refers to the act of securing data as it is stored, whether that is on disks, databases, or any other storage devices, by making it unreadable to unauthorized individuals. Protecting stored data in the event of a breach is the main idea. The policy is in place to make sure that even if hackers gain access to the storage, they won’t be able to read or use the data without the proper encryption keys. |
| Encryption at flight | This pertains to the protection of data while it is being sent over a network. Whether it's between users, servers, or applications, data is vulnerable to various cyber-attacks during transmission. The policy ensures that intercepted data remains confidential and cannot be read without decryption, which requires the right encryption keys. |
| Encryption in use | Only data that the app or software is now viewing or processing is subject to this encryption. You can do this to ensure the security of your personal information even when it's in memory or cache. This is advantageous because it protects against potential flaws or vulnerabilities that can give access to environments and programs that are currently in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is the process of confirming the legitimacy of a user, system, or application. There are several methods for doing it, including passwords, biometrics, and multi-factor authentication. It is crucial to make sure appropriate identification in order to grant access to only authorized people. In addition to protecting against illegal access, this lessens the possibility of user impersonation. |
| Authorization | Permissions establish which user, resource, or actions can be done by authenticated entities. Withholding or giving permissions according to assigned roles or standards guarantees users can only access the resources or actions relevant to their responsibilities. This policy creates a separation of responsibilities and prevents users or systems from gaining information or access to data or locations they shouldn’t. |
| Accounting | This policy involves monitoring and logging user activity on a system or network. It is crucial task that consists in keeping track of who accessed what, when, and any modifications made. It creates an audit trail of user behaviors that can be used to perform post-incident analysis. This policy is important due to being an essential piece to keeping a record for auditing purposes, ensuring the software meets compliance needs, and facilitating investigations if need be. |

**\***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 | 09/17/2023 | Listed 10 Coding Principles and Standards | Evan Bush |  |
| 3.0 | 10/08/2023 | Finalized Document and Security Policy | Evan Bush |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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