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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate all input data no matter the source. This is the first line of defense. Keep in mind potential input data sources. |
| 1. Heed Compiler Warnings | Compile code using highest warning level for compiler and eliminate reported warnings by changing the code. Additionally utilize static and dynamic tools to find additional security issues and correct them. |
| 1. Architect and Design for Security Policies | Always include implementation of security policies as part of design from the start. |
| 1. Keep It Simple | Complex code is not necessarily better code. Keep code to the simplest implementation possible to accomplish the goal. In addition to non-security reasons, the more complicated code becomes the more likely it is to contain or obfuscate security vulnerabilities. |
| 1. Default Deny | Access and permissions should always default to denial rather than defaulting to permitted. |
| 1. Adhere to the Principle of Least Privilege | Subjects should be given only those privileges that are needed in order to complete a task. Additional permissions should only be given for the duration of the task that requires them. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data before sending to other systems to remove unsafe characters. This goes hand in hand with input validation and will minimize available exploits in the event of a breach. |
| 1. Practice Defense in Depth | Always take a defense in depth approach to security by creating multiple security layers. |
| 1. Use Effective Quality Assurance Techniques | Good Quality Assurance practices should be employed in order to help in finding and removing vulnerabilities along with other issues. Some of these include Fuzz testing, penetration testing, and code audits. Whenever possible utilize external reviewers to get fresh eyes and opinions. |
| 1. Adopt a Secure Coding Standard | Select and adhere to an existing credible coding standard and/or prepare a list of coding guidelines specific for your company. |

### 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** | **Implement abstract data types using opaque types** |
| --- | --- | --- |
| **Data Type** | DCL-001-CPP | Abstract data types are not restricted to object-oriented languages such as C++ and Java. They should be created and used in C language programs as well. Abstract data types are most effective when used with private (opaque) data types and information hiding. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is based on the managed string library developed by CERT [Burch 2006]. In this example, the managed string type and the functions that operate on this type are defined in the string\_m.h header file as follows: |
| **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;    unsigned **char** strtype;  **char** \*cstr;  };    **typedef** **struct** string\_mx string\_mx;    /\* Function declarations \*/  **extern** errno\_t strcpy\_m(string\_mx \*s1, **const** string\_mx \*s2);  **extern** errno\_t strcat\_m(string\_mx \*s1, **const** string\_mx \*s2);  /\* ... \*/ |

| **Compliant Code** |
| --- |
| This compliant solution reimplements the string\_mx type as a private type, hiding the implementation of the data type from the user of the managed string library. To accomplish this, the developer of the private data type creates two header files: an external string\_m.h header file that is included by the user of the data type and an internal file that is included only in files that implement the managed string abstract data type.  In the external string\_m.h file, the string\_mx type is defined to be an instance of struct string\_mx, which in turn is declared as an incomplete type: |
| **struct** string\_mx;  **typedef** **struct** string\_mx string\_mx;    /\* Function declarations \*/  **extern** errno\_t strcpy\_m(string\_mx \*s1, **const** string\_mx \*s2);  **extern** errno\_t strcat\_m(string\_mx \*s1, **const** string\_mx \*s2);  /\* ... \*/  /\* In the internal header file, struct string\_mx is fully defined but not visible to a user of the data abstraction:\*/  **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;    unsigned **char** strtype;  **char** \*cstr;  }; |

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

| **Principles(s):** 5. Default Deny. In general, utilizing the least permissive modifier for data types defaults to denying access rather than permitting access. 6. Adhere to the principle of least privilege. Hiding the implementation of the data type from the user is an example of obfuscation in keeping with the principle of least privilege. The user is not given access to see or utilize anything but what is necessary for their task. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low | 3 |

**Automation Detection**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-DCL12** |  |
| LDRA tool suite | 9.7.1 | **104 D** | Partially implemented |
| Polyspace Bug Finder | R2022b | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |
| Parasoft C/C++test | 2022.2 | **CERT\_C-DCL12-a** | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |

**Automation**

#### Coding Standard 2

| **Coding Standard** | **Label** | **Value-returning functions must return a value from all exit paths** |
| --- | --- | --- |
| **Data Value** | MSC-002-CPP | The C++ Standard, [stmt.return], paragraph 2 [ISO/IEC 14882-2014], states the following:  Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function.  A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, such as from a function-try-block, as explained in the C++ Standard, [except.handle], paragraph 15:  Flowing off the end of a function-try-block is equivalent to a return with no value; this results in undefined behavior in a value-returning function (6.6.3). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |

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

| **Principles(s):** 4. Keep it simple. Being able to depend that all code paths for functions that return values are being implemented to return values helps keep code simple and dependable when it is potentially refactored, maintained or changed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **return-implicit** | Fully checked |
| LDRA tool suite | 9.7.1 | **2 D, 36 S** | Fully implemented |
| Parasoft C/C++test | 2022.2 | **CERT\_CPP-MSC52-a** | All exit paths from a function, except main(), with non-void return type shall have an explicit return statement with an expression |
| RuleChecker | 22.10 | **return-implicit** | Fully checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | STR-003-CPP | According to the C Standard, 6.4.5, paragraph 3 [ISO/IEC 9899:2011]:  A character string literal is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF−8 string literal is the same, except prefixed by u8. A wide string literal is the same, except prefixed by the letter L, u, or U.  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.)  Avoid assigning a string literal to a pointer to non-const or casting a string literal to a pointer to non-const. For the purposes of this rule, a pointer to (or array of) const characters must be treated as a string literal. Similarly, the returned value of the following library functions must be treated as a string literal if the first argument is a string literal:  strpbrk(), strchr(), strrchr(), strstr()  wcspbrk(), wcschr(), wcsrchr(), wcsstr()  memchr(), wmemchr()  This rule is a specific instance of EXP40-C. Do not modify constant objects. |

| **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. Attempting to modify string literals typically results in undefined behavior and oftentimes access violations. 3. Designing to avoid this behavior can avoid abnormal program termination and potential dos attacks that may result from it. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | **string-literal-modfication write-to-string-literal** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-STR30** | Fully implemented |
| Parasoft C/C++test | 2022.2 | **CERT\_C-STR30-a CERT\_C-STR30-b** | A string literal shall not be modified Do not modify string literals |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | STR-004-CPP | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted.  These are some examples of complex subsystems:  Command processor via a call to system() or similar function (also addressed in ENV03-C. Sanitize the environment when invoking external programs)  External programs  Relational databases  Third-party commercial off-the-shelf components (for example, an enterprise resource planning subsystem) |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() [Viega 2003]: |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):** 1. Sanitizing the data being passed to complex subsystems helps prevent passing potentially malicious data into those systems in keeping with validating input data. 7. Sanitizing data passed to complex systems prevents passing along data that may cause unintended functionality by those systems in keeping with the principle of sanitizing data sent to other systems. 8. Sanitizing data passed between systems is one part in creating a multi-layered defensive strategy in keeping with the principle of defense in depth. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported by stubbing/taint analysis |
| Parasoft C/C++test | 2022.2 | **CERT\_C-STR02-a CERT\_C-STR02-b CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |
| Coverity | 6.5 | **TAINTED\_STRING** | Fully implemented |
| CodeSonar | 7.2p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Free dynamically allocated memory when no longer needed** |
| --- | --- | --- |
| **Memory Protection** | MEM-005-CPP | Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to free() with that pointer value. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text\_buffer referring to the object: |
| #include <stdlib.h>    **enum** { BUFFER\_SIZE = 32 };    **int** f(**void**) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  **if** (text\_buffer == NULL) {  **return** -1;    }  **return** 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the pointer is deallocated with a call to free(): |
| #include <stdlib.h>    **enum** { BUFFER\_SIZE = 32 };    **int** f(**void**) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  **if** (text\_buffer == NULL) {  **return** -1;    }    **free**(text\_buffer);  **return** 0;  } |

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

| **Principles(s):** 3. Freeing dynamically allocated memory when no longer use is good practice for a number of reasons, but for security purposes it prevents the memory from remaining allocated and potentially exacerbating other vulnerabilities that may appear. 4. Keeping allocated memory around longer than necessary creates unnecessary complexity to track and is thus against the principle of keeping it simple. 8. For similar reasons as 3, removing this when no longer in use can help mitigate other vulnerabilities that could appear in keeping with the principle of defense in depth. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-MEM31** | Can detect dynamically allocated resources that are not freed |
| Coverity | 2017.07 | **RESOURCE\_LEAK**  **ALLOC\_FREE\_MISMATCH** | Finds resource leaks from variables that go out of scope while owning a resource |
| Parasoft C/C++test | 2022.2 | **CERT\_C-MEM31-a** | Ensure resources are freed |
| PC-lint Plus | 1.4 | **429** | Fully supported |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Understand the termination behavior of assert() and abort()** |
| --- | --- | --- |
| **Assertions** | ERR-006-CPP | The C Standard, subclause 7.2.1.1 [ISO/IEC 9899:2011], defines assert() to have the following behavior:  The assert macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if expression (which shall have a scalar type) is false (that is, compares equal to 0), the assert macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function—the latter are respectively the values of the pre-processing macros \_\_FILE\_\_ and \_\_LINE\_\_ and of the identifier \_\_func\_\_) on the standard error stream in an implementation-defined format. It then calls the abort function.  Because assert() calls abort(), cleanup functions registered with atexit() are not called. If the intention of the programmer is to properly clean up in the case of a failed assertion, then runtime assertions should be replaced with static assertions where possible. (See DCL03-C. Use a static assertion to test the value of a constant expression.) When the assertion is based on runtime data, the assert should be replaced with a runtime check that implements the adopted error strategy (see ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy).  See ERR04-C. Choose an appropriate termination strategy for more information on program termination strategies and MSC11-C. Incorporate diagnostic tests using assertions for more information on using the assert() macro. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example defines a function that is called before the program exits to clean up: |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **assert**(/\* Something bad didn't happen \*/);      /\* ... \*/  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to assert() is replaced with an if statement that calls exit() to ensure that the proper termination routines are run: |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **if** (/\* Something bad happened \*/) {  **exit**(EXIT\_FAILURE);    }      /\* ... \*/  } |

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

| **Principles(s):** 6. Proper termination routines can potentially be important in as part of a least privilege approach where access is only granted to things while it is necessary. 8. Similarly proper shut down routines potentially act a s part of defense in depth by making sure nothing that shouldn’t remaining running does so, mitigating what other vulnerabilities may gain access to or potentially closing vulnerabilities that may exist. 9. Often times additional logging occurs during proper termination routines that assist with quality assurance. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | **44 S** | Enhanced enforcement |
| Parasoft C/C++test | 2022.2 | CERT\_C-ERR06-a | Do not use assertions |
| PC-lint Plus | 1.4 | **586** | Fully supported |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Guarantee exception safety** |
| --- | --- | --- |
| **Exceptions** | ERR-007-CPP | Proper handling of errors and exceptional situations is essential for the continued correct operation of software. The preferred mechanism for reporting errors in a C++ program is exceptions rather than error codes. A number of core language facilities, including dynamic\_cast, operator new(), and typeid, report failures by throwing exceptions. In addition, the C++ standard library makes heavy use of exceptions to report several different kinds of failures. Few C++ programs manage to avoid using some of these facilities. Consequently, the vast majority of C++ programs must be prepared for exceptions to occur and must handle each appropriately. (See ERR51-CPP. Handle all exceptions.)  Because exceptions introduce code paths into a program, it is important to consider the effects of code taking such paths and to avoid any undesirable effects that might arise otherwise. Some such effects include failure to release an acquired resource, thereby introducing a leak, and failure to reestablish a class invariant after a partial update to an object or even a partial object update while maintaining all invariants. Code that avoids any such undesirable effects is said to be exception safe.  Based on the preceding effects, the following table distinguishes three kinds of exception safety guarantees from most to least desired.  Strong  The strong exception safety guarantee is a property of an operation such that, in addition to satisfying the basic exception safety guarantee, if the operation terminates by raising an exception, it has no observable effects on program state.  Strong Exception Safety  Basic  The basic exception safety guarantee is a property of an operation such that, if the operation terminates by raising an exception, it preserves program state invariants and prevents resource leaks.  Basic Exception Safety  None  Code that provides neither the strong nor basic exception safety guarantee is not exception safe.  No Exception Safety  Code that guarantees strong exception safety also guarantees basic exception safety.  Because all exceptions thrown in an application must be handled, in compliance with ERR50-CPP. Do not abruptly terminate the program, it is critical that thrown exceptions do not leave the program in an indeterminate state where invariants are violated. That is, the program must provide basic exception safety for all invariants and may choose to provide strong exception safety for some invariants. Whether exception handling is used to control the termination of the program or to recover from an exceptional situation, a violated invariant leaves the program in a state where graceful continued execution is likely to introduce security vulnerabilities. Thus, code that provides no exception safety guarantee is unsafe and must be considered defective. |

| **Noncompliant Code** |
| --- |
| The following noncompliant code example shows a flawed copy assignment operator. The implicit invariants of the class are that the array member is a valid (possibly null) pointer and that the nElems member stores the number of elements in the array pointed to by array. The function deallocates array and assigns the element counter, nElems, before allocating a new block of memory for the copy. As a result, if the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }        IntArray(**const** IntArray& that); // nontrivial copy constructor    IntArray& operator=(**const** IntArray &rhs) {  **if** (**this** != &rhs) {  **delete**[] array;        array = nullptr;        nElems = rhs.nElems;  **if** (nElems) {          array = **new** **int**[nElems];          std::**memcpy**(array, rhs.array, nElems \* **sizeof**(\*array));        }      }  **return** \***this**;    }      // ...  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the copy assignment operator provides the strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. Only after the allocation succeeds does the function proceed to change the state of the object. In addition, by copying the array to the newly allocated storage before deallocating the existing array, the function avoids the test for self-assignment, which improves the performance of the code in the common case [Sutter 2004]. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }      IntArray(**const** IntArray& that); // nontrivial copy constructor      IntArray& operator=(**const** IntArray &rhs) {  **int** \*tmp = nullptr;  **if** (rhs.nElems) {        tmp = **new** **int**[rhs.nElems];        std::**memcpy**(tmp, rhs.array, rhs.nElems \* **sizeof**(\*array));      }  **delete**[] array;      array = tmp;      nElems = rhs.nElems;  **return** \***this**;    }      // ...  }; |

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

| **Principles(s):** 9. Proper handling of errors and exception can be critical in tracking issues as part of effective quality assurance techniques. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.2 | **CERT\_CPP-ERR56-a CERT\_CPP-ERR56-b** | Always catch exceptions Do not leave 'catch' blocks empty |
| Polyspace Bug Finder | R2022b | CERT C++: ERR56-CPP | Checks for exceptions violating class invariant (rule fully covered). |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not modify the standard namespaces** |
| --- | --- | --- |
| Standard namespace | DCL-008-CPP | Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions. One feature of namespaces is that they can be further extended, even within separate translation units. For instance, the following declarations are well-formed.  namespace MyNamespace {  int length;  }    namespace MyNamespace {  int width;  }    void f() {  MyNamespace::length = MyNamespace::width = 12;  }  The standard library introduces the namespace std for standards-provided declarations such as std::string, std::vector, and std::for\_each. However, it is undefined behavior to introduce new declarations in namespace std except under special circumstances. The C++ Standard, [namespace.std], paragraphs 1 and 2 [ISO/IEC 14882-2014], states the following:  1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace std or to a namespace within namespace std unless otherwise specified. A program may add a template specialization for any standard library template to namespace std only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.  2 The behavior of a C++ program is undefined if it declares  — an explicit specialization of any member function of a standard library class template, or  — an explicit specialization of any member function template of a standard library class or class template, or  — an explicit or partial specialization of any member class template of a standard library class or class template.  In addition to restricting extensions to the the namespace std, the C++ Standard, [namespace.posix], paragraph 1, further states the following:  The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by ISO/IEC 9945 and other POSIX standards.  Do not add declarations or definitions to the standard namespaces std or posix, or to a namespace contained therein, except for a template specialization that depends on a user-defined type that meets the standard library requirements for the original template.  The Library Working Group, responsible for the wording of the Standard Library section of the C++ Standard, has an unresolved issue on the definition of user-defined type. Although the Library Working Group has no official stance on the definition [INCITS 2014], we define it to be any class, struct, union, or enum that is not defined within namespace std or a namespace contained within namespace std. Effectively, it is a user-provided type instead of a standard library–provided type. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the declaration of x is added to the namespace std, resulting in undefined behavior. |
| **namespace** std {  **int** x;  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes the intention of the programmer was to place the declaration of x into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name. |
| **namespace** nonstd {  **int** x;  } |

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

| **Principles(s):** 2. Modifying the standard namespace results in undefined behavior. 3. Standard Namespace can typically be relied upon by users for what it will contain, changing that in non-special circumstances makes designing for security more difficult. 4. For similar reasons, making what is defined in the standard namespace less reliable complicates the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.2 | **CERT\_CPP-DCL58-a** | Do not modify the standard namespaces 'std' and 'posix' |
| Polyspace Bug Finder | R2022b | CERT C++: DCL58-CPP | Checks for modification of standard namespaces (rule fully covered) |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| Unsigned integers | INT-009-CPP | The C Standard, 6.2.5, paragraph 9 [ISO/IEC 9899:2011], states  A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.  This behavior is more informally called unsigned integer wrapping. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer.  The following sections examine specific operations that are susceptible to unsigned integer wrap. When operating on integer types with less precision than int, integer promotions are applied. The usual arithmetic conversions may also be applied to (implicitly) convert operands to equivalent types before arithmetic operations are performed. Programmers should understand integer conversion rules before trying to implement secure arithmetic operations. (See INT02-C. Understand integer conversion rules.)  Integer values must not be allowed to wrap, especially if they are used in any of the following ways:  Integer operands of any pointer arithmetic, including array indexing  The assignment expression for the declaration of a variable length array  The postfix expression preceding square brackets [] or the expression in square brackets [] of a subscripted designation of an element of an array object  Function arguments of type size\_t or rsize\_t (for example, an argument to a memory allocation function)  In security-critical code  The C Standard defines arithmetic on atomic integer types as read-modify-write operations with the same representation as regular integer types. As a result, wrapping of atomic unsigned integers is identical to regular unsigned integers and should also be prevented or detected.  Addition  Addition is between two operands of arithmetic type or between a pointer to an object type and an integer type. This rule applies only to addition between two operands of arithmetic type. (See ARR37-C. Do not add or subtract an integer to a pointer to a non-array object and ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.)  Incrementing is equivalent to adding 1. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability. |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap: |
| #include <limits.h>    **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum;  **if** (UINT\_MAX - ui\_a < ui\_b) {      /\* Handle error \*/    } **else** {      usum = ui\_a + ui\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** 1. Validate Input Data. Verifying that unsigned integers do not wrap is done by validating the input data as is shown in the compliant vs non-compliant code examples. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | **integer-overflow** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-INT30** | Implemented |
| CodeSonar | 7.2p0 | **ALLOC.SIZE.ADDOFLOW** **ALLOC.SIZE.IOFLOW** **ALLOC.SIZE.MULOFLOW** **ALLOC.SIZE.SUBUFLOW** **MISC.MEM.SIZE.ADDOFLOW** **MISC.MEM.SIZE.BAD** **MISC.MEM.SIZE.MULOFLOW** **MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Coverity | 2017.07 | **INTEGER\_OVERFLOW** | Implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Predicate function objects should not be mutable** |
| --- | --- | --- |
| Predicate | CTR-010-CPP | The C++ standard library implements numerous common algorithms that accept a predicate function object. The C++ Standard, [algorithms.general], paragraph 10 [ISO/IEC 14882-2014], states the following:  [Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as reference\_wrapper<T>, or some equivalent solution. — end note]  Because it is implementation-defined whether an algorithm copies a predicate function object, any such object must either  implement a function call operator that does not mutate state related to the function object's identity, such as nonstatic data members or captured values, or  wrap the predicate function object in a std::reference\_wrapper<T> (or an equivalent solution).  Marking the function call operator as const is beneficial, but insufficient, because data members with the mutable storage class specifier may still be modified within a const member function. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to remove the third item in a container using a predicate that returns true only on its third invocation. |
| #include <algorithm>  #include <functional>  #include <iostream>  #include <iterator>  #include <vector>    **class** MutablePredicate : **public** std::unary\_function<**int**, **bool**> {  **size\_t** timesCalled;  **public**:    MutablePredicate() : timesCalled(0) {}    **bool** operator()(**const** **int** &) {  **return** ++timesCalled == 3;    }  };    **template** <**typename** Iter>  **void** print\_container(Iter b, Iter e) {    std::cout << "Contains: ";    std::copy(b, e, std::ostream\_iterator<decltype(\*b)>(std::cout, " "));    std::cout << std::endl;  }    **void** f() {    std::vector<**int**> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};    print\_container(v.begin(), v.end());      v.erase(std::remove\_if(v.begin(), v.end(), MutablePredicate()), v.end());    print\_container(v.begin(), v.end());  } |

| **Compliant Code** |
| --- |
| This compliant solution uses std::ref to wrap the predicate in a std::reference\_wrapper<T> object, ensuring that copies of the wrapper object all refer to the same underlying predicate object. |
| #include <algorithm>  #include <functional>  #include <iostream>  #include <iterator>  #include <vector>    **class** MutablePredicate : **public** std::unary\_function<**int**, **bool**> {  **size\_t** timesCalled;  **public**:    MutablePredicate() : timesCalled(0) {}    **bool** operator()(**const** **int** &) {  **return** ++timesCalled == 3;    }  };    **template** <**typename** Iter>  **void** print\_container(Iter b, Iter e) {    std::cout << "Contains: ";    std::copy(b, e, std::ostream\_iterator<decltype(\*b)>(std::cout, " "));    std::cout << std::endl;  }    **void** f() {    std::vector<**int**> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};    print\_container(v.begin(), v.end());      MutablePredicate mp;    v.erase(std::remove\_if(v.begin(), v.end(), std::ref(mp)), v.end());    print\_container(v.begin(), v.end());  } |

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

| **Principles(s):** 3. Designing early to incorporate code compliant with this makes it easier to avoid. 8. Writing code that can be relied upon to referring to the expected object can avoid opening up vulnerabilities, better securing a layer of a defense in depth approach to security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | High | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.2 | **CERT\_CPP-CTR58-a** | Make predicates const pure functions |
| Polyspace Bug Finder | R2022b | CERT C++: CTR58-CPP | Checks for function object that modifies its state (rule fully 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.

Automated enforcement of the standards encompassed by this policy should happen at code check in, at the interstitial line between design and build. Whenever new code is checked in to a separate branch before it merges to main, it should trigger automated tests to run the various tools listed in the various coding standards that have been set up as part of the check in pipeline for this purpose and raise awareness of any security policy violations. Code that does not meet the security policy requirements should fail check in compliance until it is fixed. There should also be an automated request triggered on code check in to request a peer review.

Additional automated testing should be included as part of the Verify and Test phase to retest the full build rather than focusing on newly checked in code. Some automated collecting of logs and detection can be added to the Monitor and Detection to watch for violations of these security policies occurring or being exploited.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL-001-CPP | Low | Unlikely | High | Low | 3 |
| MSC-002-CPP | Medium | Probable | Medium | Medium | 2 |
| STR-003-CPP | Low | Likely | Low | Medium | 2 |
| STR-004-CPP | High | Likely | Medium | High | 5 |
| MEM-005-CPP | Medium | Probable | Medium | Medium | 2 |
| ERR-006-CPP | Medium | Unlikely | Medium | Low | 3 |
| ERR-007-CPP | High | Likely | High | Medium | 2 |
| DCL-008-CPP | High | Unlikely | Medium | Medium | 2 |
| INT-009-CPP | High | Likely | High | Medium | 2 |
| CTR-010-CPP | Low | Likely | High | Low | 3 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 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 the encryption of stored data that is not in active use. It is typically already secured behind a firewall and general system security. We need to understand the sensitivity of the data to be secured, it should be encrypted appropriately to prevent users without authorization from being able to access and read it. A role-based accounts strategy should be utilized to allow permission to access the encrypted data and it should default to denial. Appropriate authentication security and accounting through logging of account actions are also important. |
| Encryption at flight | Encryption of data and communications as it is actively being transmitted. Encryption of the data in transit is necessary to defend against potential attacks in transit and verification of the data’s receiver is also important. |
| Encryption in use | Encryption of data in active use. This data should be obscured when reasonable, such as hashing passwords as they are entered. Access should be restricted utilizing authentication and authentication to make sure only those who should be able use the data or systems can access it. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is identifying the user and verifying that they are allowed access to the system. Typically through username, password and multifactor authentication. |
| Authorization | Authorization is deciding what permissions a legitimate user that has been authenticated should have at any given time. This is usually done through role-based accounts associated with a specific user. In principle permissions should default to denial and be the minimum necessary for the user to accomplish their goal. Furthermore, it is wise to restrict permissions to being granted only for the timeframe during which they are needed. |
| Accounting | Accounting involves tracking what actions a user takes on the system including what they have accessed, typically through logging. Accounting allows damage due to breaches to be mitigated by letting those investigating see what changes were made and what data may have been accessed. |

**\***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 | 02/11/2023 | Secure Coding Project 1 | Tim Gallus |  |
|  |  |  |  |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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