**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 | Before processing any input data, we should be verifying it to make sure it is of the desired format and range. This can help prevent common vulnerabilities like buffer overflows and SQL injection. |
| 1. Heed Compiler Warnings | This entails being aware of compiler warnings and promptly addressing them. Code faults or potential security flaws are frequently indicated by compiler warnings. |
| 1. Architect and Design for Security Policies | This includes planning software architecture and functionality to support those security policies as well as taking security requirements into account from the very beginning of the software development process. |
| 1. Keep It Simple | Reducing the possibility of introducing vulnerabilities or errors by utilizing clear, straightforward, well-understood coding methods and procedures. |
| 1. Default Deny | Automatically blocking all access and only granting it when it is expressly needed. By doing so, the attack surface is reduced, and unwanted access is avoided. This ensures that those who genuinely require access do so by going through the appropriate routes. |
| 1. Adhere to the Principle of Least Privilege | Like default deny, provide processes and users with the minimal level of access required for them to execute their tasks. Limiting the amount of damage an attacker can cause can help to lessen the effects of a successful attack by having less information available. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing all data that is sent to other systems, such as databases or web services. This can help to prevent vulnerabilities such as SQL injection or cross-site scripting. |
| 1. Practice Defense in Depth | Putting in place many layers of security measures to defend against various assaults. Most systems need more than one layer of defense; by combining levels of defense, you can prevent a system from being breached and causing havoc. |
| 1. Use Effective Quality Assurance Techniques | Software must be rigorously tested and verified to guarantee that it is safe and resistant to attacks. This can involve methods like penetration and unit testing. Systems can be made more secure by employing numerous testing phases, independent security assessments, and external security reviews. |
| 1. Adopt a Secure Coding Standard | This requires following recognized secure code standards and recommendations, such as those made public by groups like OWASP or CERT. By doing this, software development best practices can be followed, and security concerns can be reduced. |

### 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** | [STD-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);  /\* ... \*/ |

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

| **Principles(s):** 4. Keep It Simple: Software design and implementation should be kept as simple as possible to minimize potential security vulnerabilities.  9. Use Effective Quality Assurance Techniques: We need testing and code review to ensure that the use of opaque types does not introduce new vulnerabilities or other issues.  10. Adopt a Secure Coding Standard: The standard itself is a secure coding standard for implementing abstract data types using opaque types. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | P3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 104 D | Partially implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | 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](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 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 |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Value-returning functions must return a value from all exit paths** |
| --- | --- | --- |
| **Data Value** | [STD-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):** 1. Validate Input Data: this ensuring that any input data passed to value-returning functions is valid and within acceptable ranges. This helps prevent unexpected behavior or security issues caused by invalid input data.  9. Use Effective Quality Assurance Techniques: quality assurance techniques such as code reviews and testing can help ensure that all value-returning functions in the codebase return a value from all exit paths.  10. Adopt a Secure Coding Standard: This promotes the use of coding standards that require developers to follow specific guidelines for writing code that ensures all code paths in a value-returning function have a return statement. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | return-implicit | Fully checked |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 2 D, 36 S | Fully implemented |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as function-try-blocks |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | LANG.STRUCT.MRS | Missing return statement |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Use valid format strings** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP]] | The formatted output functions (fprintf() and related functions) convert, format, and print their arguments under control of a format string. |

| **Noncompliant Code** |
| --- |
| Mismatches between arguments and conversion specifications may result in undefined behavior. Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the error\_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. Likewise, the error\_msg argument is incorrectly matched with the d specifier instead of the s specifier. These usages result in undefined behavior. One possible result of this invocation is that printf() will interpret the error\_type argument as a pointer and try to read a string from the address that error\_type contains, possibly resulting in an access violation. |
| #include <stdio.h>    void func(void) {  const char \*error\_msg = "Resource not available to user.";  int error\_type = 3;  /\* ... \*/  printf("Error (type %s): %d\n", error\_type, error\_msg);  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications: |
| #include <stdio.h>    void func(void) {  const char \*error\_msg = "Resource not available to user.";  int error\_type = 3;  /\* ... \*/  printf("Error (type %d): %s\n", error\_type, error\_msg);    /\* ... \*/  } |

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

| **Principles(s):** 1. Validate Input Data: By validating input data, we can ensure that the data is in the correct format and type to be used with the format string.  10. Adopt a Secure Coding Standard: Have guidelines for how to properly use format strings, preventing vulnerabilities caused by incorrect use of these strings. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | 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-FIO47 | Fully implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | IO.INJ.FMT  MISC.FMT  MISC.FMTTYPE | Format string injection  Format string  Format string type error |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | PW | Reports when the number of arguments differs from the number of required arguments according to the format string |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | match format and arguments | Exhaustively verified (see the compliant and the non-compliant example). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-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. |

| **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() |
| 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. Validate Input Data: Sanitizing data is one way of validating input data, and it helps to ensure that the data passed to complex subsystems is safe and secure.  6. Adhere to the Principle of Least Privilege: The potential damage that can be caused by malicious data sent to complex subsystems can be minimized by preventing unauthorized access or modification of data.  7. Sanitize Data Sent to Other Systems: By sanitizing data passed to complex subsystems, the risk of attacks such as injection attacks can be mitigated. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [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. |
| [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 |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Only free memory allocated dynamically** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP]] | Freeing memory that is not allocated dynamically can result in heap corruption and other serious errors. Do not call free() on a pointer other than one returned by a standard memory allocation function, such as malloc(), calloc(), realloc(), or aligned\_alloc(). |

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

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

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

| **Principles(s):** 1. Validate Input Data: To verify that memory being freed was allocated dynamically, and not statically or automatically. Validation can prevent errors such as freeing memory that was not dynamically allocated or freeing memory multiple times.  9. Use Effective Quality Assurance Techniques: Quality assurance techniques such as code reviews and testing can help ensure that memory is being freed correctly and only when it should be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.04 | invalid-free | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | ALLOC.TM | Type Mismatch |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | BAD\_FREE | Identifies calls to free() where the argument is a pointer to a function or an array. It also detects the cases where  free() is used on an address-of expression, which can never be heap allocated. Coverity Prevent cannot discover all  violations of this rule, so further verification is necessary |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C: Rule MEM34-C | Checks for:  Invalid free of pointer  Invalid reallocation of pointer  Rule fully covered |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Incorporate diagnostic tests using assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP]] | When it is executed, if expression (which must have a scalar type) is false, the assert macro outputs information about the failed assertion (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function) on the standard error stream, in an implementation-defined format, and calls the abort() function. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char \*)malloc(len + 1);  assert(NULL != dup);    memcpy(dup, c\_str, len + 1);  return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char\*)malloc(len + 1);  /\* Detect and handle memory allocation error \*/  if (NULL == dup) {  return NULL;  }    memcpy(dup, c\_str, len + 1);  return dup;  } |

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

| **Principles(s):** 9. Use Effective Quality Assurance Techniques: Incorporating diagnostic tests using assertions is a quality assurance technique that can help detect errors and vulnerabilities in the code.  10. Adopt a Secure Coding Standard: Incorporating diagnostic tests using assertions in the code to identify and handle error conditions can help ensure that the code meets secure coding practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Honor exception specifications** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP]] | If a function throws an exception other than one allowed by its exception-specification, it can lead to an implementation-defined termination of the program  If a function declared with a dynamic-exception-specification throws an exception of a type that would not match the exception-specification, the function std::unexpected() is called. The behavior of this function can be overridden but, by default, causes an exception of std::bad\_exception to be thrown. Unless std::bad\_exception is listed in the exception-specification, the function std::terminate() will be called. |

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

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

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

| **Principles(s):** 1. Validate Input Data: By validating input data, the code can ensure that it adheres to the exception specifications, helping to prevent violations.  9. Use Effective Quality Assurance Techniques: Code can be tested for exceptions and other issues before it is released, helping to prevent security issues and other problems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | unhandled-throw-noexcept | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | LANG.STRUCT.EXCP.THROW | Use of throw |
| [Parasoft C/C++Test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-ERR55-a | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | CERT C++: ERR55-CPP | Checks for noexcept functions exiting with exception (rule fully covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not call a function with a mismatched language linkage** |
| --- | --- | --- |
| Language | [STD-008-CPP]] | Do not call a function through a type whose language linkage does not match the language linkage of the called function's definition. This restriction applies both to functions called within a C++ program as well as function pointers used to make a function call from outside of the C++ program. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call\_java\_fn\_ptr() function expects to receive a function pointer with "java" language linkage because that function pointer will be used by a Java interpreter to call back into the C++ code. However, the function is given a pointer with "C++" language linkage instead, resulting in undefined behavior when the interpreter attempts to call the function pointer. This code should be ill-formed because the type of callback\_func() is different than the type java\_callback. However, due to common implementation divergence from the C++ Standard, some compilers may incorrectly accept this code without issuing a diagnostic. |
| extern "java" typedef void (\*java\_callback)(int);    extern void call\_java\_fn\_ptr(java\_callback callback);  void callback\_func(int);    void f() {  call\_java\_fn\_ptr(callback\_func);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the callback\_func() function is given "java" language linkage to match the language linkage for java\_callback. |
| extern "java" typedef void (\*java\_callback)(int);    extern void call\_java\_fn\_ptr(java\_callback callback);  extern "java" void callback\_func(int);    void f() {  call\_java\_fn\_ptr(callback\_func);  } |

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

| **Principles(s):** 10. Adopt a Secure Coding Standard: This helps ensure that all functions in the code are written with the appropriate language linkage consistently throughout the development process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-EXP56-a | Do not call a function with a mismatched language linkage |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not rely on the value of a moved-from object** |
| --- | --- | --- |
| Objects | [STD-009-CPP]] | Do not rely on the value of a moved-from object unless the type of the object is documented to be in a well-specified state. While the object is guaranteed to be in a valid state, relying on unspecified values leads to unspecified behavior. Since the behavior need not be documented, this can in turn result in abnormal program behavior and portability concerns. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the integer values 0 through 9 are expected to be printed to the standard output stream from a std::string rvalue reference. However, because the object is moved and then reused under the assumption its internal state has been cleared, unexpected output may occur despite not triggering undefined behavior. |
| #include <iostream>  #include <string>    void g(std::string v) {  std::cout << v << std::endl;  }    void f() {  std::string s;  for (unsigned i = 0; i < 10; ++i) {  s.append(1, static\_cast<char>('0' + i));  g(std::move(s));  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::string object is initialized to the expected value on each iteration of the loop. This practice ensures that the object is in a valid, specified state prior to attempting to access it in g(), resulting in the expected output. |
| #include <iostream>  #include <string>    void g(std::string v) {  std::cout << v << std::endl;  }    void f() {  for (unsigned i = 0; i < 10; ++i) {  std::string s(1, static\_cast<char>('0' + i));  g(std::move(s));  }  } |

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

| **Principles(s):** 4. Keep It Simple: Moving from an object can make it challenging to reason about the state of the object, increasing the complexity of the code. The principle of keeping it simple recommends reducing complexity wherever possible.  9. Use Effective Quality Assurance Techniques: Proper testing can identify issues related to moved-from objects, such as undefined behavior or data corruption, and ensure that the code is behaving as expected.  10. Adopt a Secure Coding Standard: Relying on the value of a moved-from object can result in security vulnerabilities, such as use-after-free errors. Adhering to a secure coding standard can help identify and prevent such issues. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | LANG.MEM.NPD  LANG.MEM.UVAR | Null Pointer Dereference  Uninitialized Variable |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-EXP63-a | Do not rely on the value of a moved-from object |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | CERT C++: EXP63-CPP | Checks for read operations that reads the value of a moved-from object (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not cast or delete pointers to incomplete classes** |
| --- | --- | --- |
| Pointers | [STD-010-CPP]] | Referring to objects of incomplete class type, also known as forward declarations, is a common practice. One such common usage is with the "pimpl idiom" [Sutter 00] whereby an opaque pointer is used to hide implementation details from a public-facing API. However, attempting to delete a pointer to an object of incomplete class type can lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a class attempts to implement the pimpl idiom but deletes a pointer to an incomplete class type, resulting in undefined behavior if Body has a nontrivial destructor. |
| class Handle {  class Body \*impl; // Declaration of a pointer to an incomplete class  public:  ~Handle() { delete impl; } // Deletion of pointer to an incomplete class  // ...  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the deletion of impl is moved to a part of the code where Body is defined. |
| class Handle {  class Body \*impl; // Declaration of a pointer to an incomplete class  public:  ~Handle();  // ...  };    // Elsewhere  class Body { /\* ... \*/ };    Handle::~Handle() {  delete impl;  } |

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

| **Principles(s):** 6. Adhere to the Principle of Least Privilege: A program should not have access to an incomplete object if it does not require access to it.  9. Use Effective Quality Assurance Techniques: The software should be thoroughly tested to ensure that there are no issues related to casting or deleting pointers to incomplete classes.  10. Adopt a Secure Coding Standard: A secure coding standard can provide guidance on how to avoid casting or deleting pointers to incomplete classes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Coverity) | 6.5 | DELETE\_VOID | Fully implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | LANG.CAST.PC.INC | Conversion: pointer to incomplete |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | CERT\_CPP-EXP57-a  CERT\_CPP-EXP57-b | Do not delete objects with incomplete class at the point of deletion  Conversions shall not be performed between a pointer to an incomplete type and any other type |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | CERT C++: EXP57-CPP | Checks for conversion or deletion of incomplete class pointer |

### 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.

When writing code, you should utilize automated testing to create a program with the main goal to protect the project from mistakes and warnings. We should use automated testing to check the programmer's work. Code snippets and project areas can be automatically checked by a software with minimal human input thanks to the DevOps process automation. By reviewing and testing the code before it is permitted to be pushed into the main branch, the automated process will safeguard the project's main branch. These tests shield the project from faulty and insufficient pieces of code that can endanger the main project's security. Each team member's developed code can be subjected to the same testing and standards thanks to an automation software.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | High | 1 | 3 |
| STD-002-CPP | Medium | Probable | Medium | 8 | 2 |
| STD-003-CPP | High | Unlikely | Medium | 6 | 2 |
| STD-004-CPP | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Unlikely | High | 1 | 3 |
| STD-007-CPP | Low | Likely | Low | 9 | 2 |
| STD-008-CPP | Low | Unlikely | Medium | 2 | 3 |
| STD-009-CPP | Medium | Probable | Medium | 8 | 2 |
| STD-010-CPP | Medium | Unlikely | Medium | 4 | 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 in rest refers to encrypting data when it is stored in a database or on a disk. This type of encryption protects the data from unauthorized access in case the device is stolen or hacked. The policy for encryption in rest should apply to all sensitive data that is stored on devices or databases. |
| Encryption at flight | Encryption in flight refers to encrypting data when it is being transmitted over a network. This type of encryption protects the data from interception and unauthorized access during transmission. The policy for encryption in flight should apply to all sensitive data that is transmitted over public or unsecured networks. |
| Encryption in use | Encryption in use refers to encrypting data while it is being processed or used by an application or service. This type of encryption protects the data from unauthorized access by applications or services that do not have permission to access the data. The policy for encryption in use should apply to all sensitive data that is processed or used by applications or services. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of confirming a user's or system's identification. Credentials like usernames and passwords, biometric information, or security tokens are used for this. All systems and programs that demand user authentication should be subject to the authentication policy. A powerful authentication technique, such as two-factor authentication or biometric authentication, must be used to verify the identity of all users. Passwords must be complicated and changed frequently. The policy and procedures for authentication must be taught to all employees and contractors who have access to the systems or applications. |
| Authorization | Authorization is the process of granting access to a user or system based on their identity and the permissions they have been granted. This guarantees that systems and users can only access the information and resources to which they have been granted access. All programs and systems that demand user authorization should be subject to the authorization policy. The least privilege principle must be followed when granting access to information and resources. Only the information and resources that users and systems actually need to carry out their tasks must be made available to them. When not needed, access must be periodically checked and revoked. The policies and procedures for authorization must be taught to all employees and contractors who have access to the systems or applications. |
| Accounting | Accounting is the process of tracking and auditing user and system activity. This makes it possible to identify and look into any unwanted access or activity. All applications and systems that call for the monitoring and tracking of user and system activities should be subject to the accounting policy. Logs of user and system activity must be created and kept up to date by all systems and applications. Regular log reviews are required, and any unusual activity needs to be looked at. The logs must only be accessible to authorized individuals. The policy and procedures for accounting must be taught to all employees and contractors who have access to the systems or apps. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 03/18/2023 | Module 3 Milestone | Brie Carlson |  |
| 3.0 | 04/04/2023 | Project One | Brie Carlson |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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