**Green Pace Developer: Jacob McPherson**



# 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 | Double check all input being supplied to your database or server. This can include checking for faulty inputs, command phrases, or overflows that could cause your software to crash. By validating the input, you can prevent a majority of software vulnerabilities. |
| 1. Heed Compiler Warnings | When compiling your code, you should have warning levels set to maximum. When a warning is sent out you should review the code around the error and modify your code accordingly. Compiler warnings are one of your first warning signs of bad code. |
| 1. Architect and Design for Security Policies | When designing your code, implement security policies throughout to ensure overall security of the program. This can be as simple as grouping subsystems together that require similar security measures. Working with smaller groups of your code can also make refining security easier. |
| 1. Keep It Simple | Keep the design of your code as simple as possible. The more complexity you add to your code the more possibility you leave for errors and input mishandling. Another thing to note is that the complexity of the security mechanisms will also determine the complexity of the minimum secure code. Simplicity can allow for easy to manage and read code when managed properly. |
| 1. Default Deny | Don’t design your code to keep users excluded from a function based on conditions. Instead design it to where no one has access, and they must provide correct information to prove they have permission to use the given function. This also means that you shouldn’t design functions to look for reasons not to do something. |
| 1. Adhere to the Principle of Least Privilege | Any function or operation that must be undertaken by the program should be accomplished with minimum permissions. Functions that provide greater permissions for a short time should ensure that permission elevation is only allowed for a very short time. Keeping an eye on this can help prevent the execution of arbitrary code due to privilege modification. |
| 1. Sanitize Data Sent to Other Systems | Information that must be passed to a more complex subgroup of your system should be sanitized before any calculations are done. This is similar in nature to validating your input; however, this focuses more on preventing attackers from accessing other potential functionality of your methods. Sanitization ensures that whatever input goes through will produce valid results that can be used by the rest of the system. |
| 1. Practice Defense in Depth | Try to control the number of threats and odd inputs that make it into your program by setting up multiple layers of defense throughout your system. Each layer of security can look for different flags or prevent different manners of input from proceeding further into method calls or database access. Different layers may also use different security techniques to accomplish their goals. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques should be used throughout your coding process. Penetration testing, input tests, and other quality assurance techniques can greatly improve the security of your code by bringing to light potential issues. Having outside individuals review your source code can also lead to more secure code in the end. Bringing in a fresh pair of eyes can identify issues that someone working on the code may overlook. |
| 1. Adopt a Secure Coding Standard | This is a little more self-explanatory. It is extremely recommended that you find or develop a standard for coding that is secure and centered around your target development language. This standard will not only give you a guideline for how to design your program, but it will allow anyone who comes in to work on your code to adhere to a defined and accepted standard that is consistent throughout the program. Without a standard you could end up with a program that looks like a patchwork project with no consideration for its’ separate functionalities. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-INT32-C] | Ensure that operations on signed integers do not result in overflow. |

| **Noncompliant Code** |
| --- |
| This example has an addition operation that could very easily be overflowed if the values provided were very high. This could lead to an incorrect value or program termination. |
| The second example shows a subtraction overflow possibility. |
| void func(signed int num1, signed int num2) {  signed int sum = num1 + num2;  cout << sum << endl;  } |
| void func(signed int si\_a, signed int si\_b) {  signed int diff = si\_a - si\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This example corrects the above error by checking to see if the provided second value will push the first value past the upper or lower maximum. |
| The second example adds a check to ensure no overflow can occur after subtraction. |
| #include <limits.h>    void func(signed int num1, signed int num2) {  signed int sum;  if (((num2 > 0) && (num1 > (INT\_MAX - num2))) ||  ((num2 < 0) && (num1 < (INT\_MIN - num2)))) {  Cout << “overflow detected. Function has been exited.” << endl;  /\* exit function as desired \*/  } else {  sum = num1 + num2;  }  cout << sum << endl;  } |
| #include <limits.h>    void func(signed int si\_a, signed int si\_b) {  signed int diff;  if ((si\_b > 0 && si\_a < INT\_MIN + si\_b) ||  (si\_b < 0 && si\_a > INT\_MAX + si\_b)) {  /\* Handle error \*/  } else {  diff = si\_a - si\_b;  }    /\* ... \*/  } |

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

| **Principles(s):** This standard follows principle number 1 almost to the letter. This standard focuses on validating values, some of which could be input by users aiming to cause an overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | Integer-overflow | Fully checked |
| Helix QAC | 2022.4 | **C2800, C2860**  **C++2800, C++2860**  **DF2801, DF2802, DF2803, DF2861, DF2862, DF2863** | [Insert text.] |
| Coverity | 2017.07 | **TAINTED\_SCALAR**  **BAD\_SHIFT** | Implemented |
| PVS-Studio | 7.23 | [**V1026**](https://pvs-studio.com/en/docs/warnings/v1026/)**,** [**V1070**](https://pvs-studio.com/en/docs/warnings/v1070/)**,** [**V1081**](https://pvs-studio.com/en/docs/warnings/v1081/)**,** [**V1083**](https://pvs-studio.com/en/docs/warnings/v1083/)**,** [**V1085**](https://pvs-studio.com/en/docs/warnings/v1085/)**,** [**V5010**](https://pvs-studio.com/en/docs/warnings/v5010/) | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-EXP34-C] | Do not dereference null pointers |

| **Noncompliant Code** |
| --- |
| When initializing size if input\_str is 0 you would end up referencing a null pointer. The initilization of c\_str could subsequently be a null pointer if for any reason size == 0. The memcpy() function would then dereference both null pointers. |
| The second example uses the libpng library on a popular cellphone that changes malloc() to return a nullpointer on error or being passed a 0. |
| #include <string.h>  #include <stdlib.h>    void func(const char \*input\_str) {  size\_t size = strlen(input\_str) + 1;  char \*c\_str = (char \*)malloc(size);  memcpy(c\_str, input\_str, size);  /\* size could possibly be null along with c\_str \*/  free(c\_str);  c\_str = NULL;  return;  } |
| #include <png.h> /\* From libpng \*/  #include <string.h>    void func(png\_structp png\_ptr, int length, const void \*user\_data) {  png\_charp chunkdata;  chunkdata = (png\_charp)png\_malloc(png\_ptr, length + 1);  /\* ... \*/  memcpy(chunkdata, user\_data, length);  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This example checks both variables to ensure they are not null before they are used in functions. Should either variable be NULL, the function exits with an error message. |
| The second example checks the pointer and uses size\_t to pass length so no negatives are passed to func(). |
| #include <string.h>  #include <stdlib.h>    void func(const char \*input\_str) {  size\_t size;  char \*c\_str;    if (input\_str == NULL) {  cout << “null pointer created. Exiting function” << endl;  return;  }    size = strlen(input\_str) + 1;  c\_str = (char \*)malloc(size);    if (c\_str == NULL) {  cout << “null pointer created. Exiting function” << endl;  return;  }    memcpy(c\_str, input\_str, size);  /\* ... \*/  free(c\_str);  c\_str = NULL;  return;  } |
| #include <png.h> /\* From libpng \*/  #include <string.h>    void func(png\_structp png\_ptr, size\_t length, const void \*user\_data) {  png\_charp chunkdata;  if (length == SIZE\_MAX) {  /\* Handle error \*/  }  if (NULL == user\_data) {  /\* Handle error \*/  }  chunkdata = (png\_charp)png\_malloc(png\_ptr, length + 1);  if (NULL == chunkdata) {  /\* Handle error \*/  }  /\* ... \*/  memcpy(chunkdata, user\_data, length);  /\* ... \*/    } |

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

| **Principles(s):** Principle 1 encapsulates the purpose of this standard. When working with ever changing variables you want to validate that you’re passing a value and not trying to reference nothing. Validating your references before you use them can save you hours of debugging. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube C/C++ Plugin | 3.11 | [**S2259**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-2259) | [Insert text.] |
| PRQA QA-C | 9.7 | **2810, 2811, 2812, 2813** | Fully implemented |
| Helix QAC | 2022.4 | **DF2810, DF2811, DF2812, DF2813** | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-STR32-C] | Do not pass a non-null-terminated character sequence to a library function that expects a string. |

| **Noncompliant Code** |
| --- |
| In this example a character array is created with 7 bits available. The provided string is exactly 7 bits long and therefore takes the entirety of the array. This means that no terminator could be appended to the end. |
| In the second example if no null terminator is in the allotted characters, then the resulting c\_str is non-null terminated. |
| #include <stdio.h>    void func(void) {  char c\_str[7] = “Hi guys”; // hi guys will not have a terminator at the end.  printf(“%s\n”, c\_str);  } |
| #include <string.h>    enum { STR\_SIZE = 32 };    size\_t func(const char \*source) {  char c\_str[STR\_SIZE];  size\_t ret = 0;    if (source) {  c\_str[sizeof(c\_str) - 1] = '\0';  strncpy(c\_str, source, sizeof(c\_str));  ret = strlen(c\_str);  } else {  /\* Handle null pointer \*/  }  return ret;  } |

| **Compliant Code** |
| --- |
| This example corrects the above error by not designating the allocation size manually. This allows the array to allocate exactly enough space for the string and terminator. This could mean other memory is accessed along with the string provided when print is called. |
| The second example corrects the error if the purpose is to truncate the string. |
| #include <stdio.h>    void func(void) {  char c\_str[] = “Hello again”;  printf(“%s\n”, c\_str);  } |
| #include <string.h>    enum { STR\_SIZE = 32 };    size\_t func(const char \*source) {  char c\_str[STR\_SIZE];  size\_t ret = 0;    if (source) {  strncpy(c\_str, source, sizeof(c\_str) - 1);  c\_str[sizeof(c\_str) - 1] = '\0';  ret = strlen(c\_str);  } else {  /\* Handle null pointer \*/  }  return ret;  } |

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

| **Principles(s):** Principles 1 and 4 cover the basics of this standard. Principle 1 comes into play when you create validation checks for your strings to ensure they are formatted properly. Principle 4 pertains to removing the preallocated size of the array variable. Using a simple open array that can allocate as necessary helps to prevent excess memory accessing. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | **STRING\_NULL** | Fully implemented |
| Helix QAC | 2022.4 | **DF2835, DF2836, DF2839** | [Insert text.] |
| PRQA QA-C | 9.7 | **2835, 2836, 2839** | [Insert text.] |
| PVS-Studio | 7.23 | [**V692**](https://pvs-studio.com/en/docs/warnings/v692/) | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-FIO30-C] | Exclude user input from format strings |

| **Noncompliant Code** |
| --- |
| In this example the user input is being added into the format created with msg\_format. If a user were to provide malicious input, you could end up with a format error or arbitrary code execution. |
| The second example uses POSIX syslog() which is also susceptible to format-string vulnerabilities. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = “%s cannot be authenticated.\n”;  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* no memory allocated. Exit function \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  fprintf(stderr, msg);  free(msg);  } |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <syslog.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  syslog(LOG\_INFO, msg);  free(msg);  } |

| **Compliant Code** |
| --- |
| This example changes the last function from fprintf() to fputs() in order to directly put out the user input without executing it fully. This helps prevent user input from being executed in the event it is malicious. |
| The second one corrects the issue by passing user input as an argument to syslog() instead. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = “%s cannot be authenticated.\n”;  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  fputs(msg, stderr);  free(msg);  } |
| #include <syslog.h>    void incorrect\_password(const char \*user) {  static const char msg\_format[] = "%s cannot be authenticated.\n";  syslog(LOG\_INFO, msg\_format, user);  } |

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

| **Principles(s):** Principles 6 and 7 are the basis for this standard. Principle 6 pertains to making sure the user input can only accomplish its given task and not allow malicious input to manipulate program output. Principle 7 in this standard relates to checking the variable that is being used for its origin (such as user input) and taking the necessary steps to ensure its validity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported via stubbing/taint analysis |
| Klocwork | 2022.4 | **SV.FMTSTR.GENERIC** **SV.TAINTED.FMTSTR** | [Insert text.] |
| CodeSonar | 7.2p0 | **IO.INJ.FMT** **MISC.FMT** | Format string injection Format string |
| PVS-Studio | 7.23 | [**V618**](https://pvs-studio.com/en/docs/warnings/v618/) | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-MEM51-CPP] | Properly deallocate dynamically allocated resources. |

| **Noncompliant Code** |
| --- |
| In this example we see that the structure S is created with the new S() operater, however it is attempted to be freed using the standard free() operator. This results in memory not being deallocated and undefined behavior. |
| The second example has variable space created by placement new operator. We then try to deallocate using the operator delete(). This results in undefined behavior. |
| #include <cstdlib>    struct S {  ~S();  };    void f() {  S \*s = new S();  // ...  std::free(s);  } |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    delete s1;  } |

| **Compliant Code** |
| --- |
| Here we correct the above error by deallocating with the delete destructor. |
| The correction to the second example is removing the call to delete() and instead invoking the structures destructor. |
| struct S {  ~S();  };    void f() {  S \*s = new S();  // ...  delete s;  } |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    s1->~S();  } |

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

| **Principles(s):** Principle 1 is the core idea behind this standard. When allocating any resource for your software you must take care and ensure you use the proper terminator for the allocation method you’ve chosen. Validating that memory has been deallocated would identify any mistakes in resource deallocation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **invalid\_dynamic\_memory\_allocation** **dangling\_pointer\_use** | [Insert text.] |
| CodeSonar | 7.2p0 | **ALLOC.FNH** **ALLOC.DF** **ALLOC.TM** **ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |
| PRQA QA-C++ | 4.4 | **2110, 2111, 2112, 2113, 2118,**  **3337, 3339, 4262, 4263, 4264** | [Insert text.] |
| PVS-Studio | 7.23 | [**V515**](https://pvs-studio.com/en/docs/warnings/v515/), [**V554**](https://pvs-studio.com/en/docs/warnings/v554/), [**V611**](https://pvs-studio.com/en/docs/warnings/v611/), [**V701**](https://pvs-studio.com/en/docs/warnings/v701/), [**V748**](https://pvs-studio.com/en/docs/warnings/v748/), [**V773**](https://pvs-studio.com/en/docs/warnings/v773/), [**V1066**](https://pvs-studio.com/en/docs/warnings/v1066/) | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-AST1-CPP] | Do not use assertions for basic error handling. |

| **Noncompliant Code** |
| --- |
| The assert() function in this example is used towards the end of function main to check to see if x has been altered. In the event x is altered in any way the program will terminate with an error message and the last section of main will never be reached. While mundane should the program need to continue and only catch the error the assertion will instead always terminate the program. |
|  |
| #include <assert.h>  Int main(void){  Int x = 14;  /\* potential modification \*/  assert(x == 14);  Int total = x + 20;  return 0;  } |

| **Compliant Code** |
| --- |
| This example moves the assertion to the beginning of main as a termination check to ensure x is properly set in the beginning before any potential calculations are done. This complies since it is done at the beginning of the program where variables are being initialized. |
|  |
| #include <assert.h>  Int main(void){  Int x = 14;  assert(x == 14);  /\* continued modification or use \*/  Int total = x + 20;  return 0;  } |

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

| **Principles(s): Principles 3 and 9 explain the reasoning for this standard the most. Principle 3 discusses designing your code with security in mind and prepping for those measures in place. With the correction of the above code we can see how the assertion aids in security checks for variables. Principle 9 would pertain more to the techniques and strategies put in place to catch such assertion errors. Testing and reviewing your code blocks** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-ERR51-CPP] | Handle all exceptions. |

| **Noncompliant Code** |
| --- |
| This example has a function that has the potential to throw exceptions, however there are no catch blocks to handle these thrown exceptions. This will lead to termination since no catch can be found. |
| In the second example the function thread\_start() does not catch exceptions. Should an exception be thrown then std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start() {  throwing\_func();  }    void f() {  std::thread t(thread\_start);  t.join();  } |

| **Compliant Code** |
| --- |
| This example fixes the above issue by encasing f() in a try block followed by a catch block to handle any exceptions thrown. This ensure the program does not terminate suddenly without the user understanding. |
| The second example is corrected by adding a catch function to handle any exceptions thrown. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start(void) {  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    void f() {  std::thread t(thread\_start);  t.join();  } |

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

| **Principles(s):** Principles 3, 8, and 9 all pertain to this standard of coding. Designing your code with try and catch blocks with their respective error handling and creating throw scenarios for wrong value inputs follows principles 3 and 8 pretty closely. Spreading try and catch blocks throughout your code where variables are being manipulated follows principle 9. |
| --- |

**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 |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.4 | **C++4035, C++4036, C++4037** | [Insert text.] |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022b | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File handling | [STD-FIO51-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| In this example you can see that a file is opened and then assumed to be closed at termination, however the termination call does not subsequently call destructors. This could allow a hacker to slowly eat up system resources and potentially crash the whole system. |
|  |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| This example closes the opened file stream with a call to file.close() and checks to see if it failed or not. |
|  |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):** Ensuring your files are closed before moving forward in your code is directly related to principles 1 and 9. When using any filestreaming functionality you must validate that you have closed your file when finished or else it could be overwritten by future code. This then ties in to principle 9 since regular checks on file development and output would ensure your files don’t fall victim to corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.4 | **DF4786, DF4787, DF4788** | [Insert text.] |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-FIO51-a** | [Insert text.] |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022b | [CERT C++: FIO51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio51cpp.html) | Checks for resource leak (rule partially covered) |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error handling | [STD-ERR50-CPP] | Do not abruptly terminate the program. |

| **Noncompliant Code** |
| --- |
| In this example the call to f() as an exit handler may cause a termination call should the throwing\_func() throw an exception when called. |
|  |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| This example prevents the termination call by creating a try catch block in function f() to handle any possible exceptions thrown by throwing\_func(). |
|  |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):** This standard follows principles 8, 9, and 10 as they all center around building the code to handle threats. Principle 8 would focus on the layered security part, while 9 and 10 relate to checks and tests to ensure errors will not occur that could cause a software crash. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **122 S** | Enhanced Enforcement |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022b | [CERT C++: ERR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr50cpp.html) | Checks for implicit call to terminate() function (rule partially covered) |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.23 | [**V667**](https://pvs-studio.com/en/docs/warnings/v667/)**,** [**V2014**](https://pvs-studio.com/en/docs/warnings/v2014/) | [Insert text.] |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S990**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-990) | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declaration safety | [STD-DCL52-CPP] | Never qualify a reference type with const or volatile. |

| **Noncompliant Code** |
| --- |
| This example has a constant reference to p created that still registers as being assignable. This can lead to undefined behavior or wrong output on some compilers. |
|  |
| #include <iostream>    void f(char c) {  char &const p = c;  p = ‘p’;  std::cout << p << std::endl;  } |

| **Compliant Code** |
| --- |
| Here we remove the const call and just create a proper reference that can then be assigned if we need it to be as shown below. |
|  |
| #include <iostream>    void f(char c) {  char &p = c;  p = ‘p’;  std::cout << p << std::endl;  } |

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

| **Principles(s):** Going over your pointer variables and ensuring they aren’t accidently constants (or attempted to be) follows a little bit of principles 3 and 9. Principle 3 is the structure for how the code is built to work and operate. Principle 9 focuses on quality assurance, which this standard would require the developer to put checks in place to ensure references returned their correct values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL52** | [Insert text.] |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022b | [CERT C++: DCL52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl52cpp.html) | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S3708**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-3708) | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

Green Pace’s current DevOps process has a very well establish response cycle for development and analysis, however I could see benefit to them adding in automated standard testing shortly after the building process is complete. Many of the standards in this document circulate around variable testing and error handling. These factors can be tested with temporary inputs shortly after they’re developed to ensure they handle basic situations. The automated tests would also allow for a smoother transition into the vulnerability and security testing phase of the DevOps process. Should any of the automated tests come back with error reports, developers will be able to move back into the building phase of the DevOps process without having to pull your testing personnel in only to find the same issue manually.

### 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-AST1-CPP | Medium | Probable | Remediation | P8 | L2 |
| STD-DCL52-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-ERR50-CPP | Low | Probable | Medium | P4 | L3 |
| STD-ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| STD-EXP34-C | High | Likely | Medium | P18 | L1 |
| STD-FIO30-C | High | Likely | Medium | P18 | L1 |
| STD-FIO51-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-INT32-C | High | Likely | High | P9 | L2 |
| STD-MEM51-CPP | High | Likely | Medium | P18 | L1 |
| STD-STR32-C | High | Probable | Medium | P12 | L1 |

### 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 the process of encrypting data and information that has been stored away and is not being accessed by anyone for long stretches of time. While we can put up firewalls and block out unauthorized users, we don’t want the information stored away to be easily read. Should all of our security measures fail it is important for the information stored away to be in a form that should be useless to an attacker. Encrypting the information while it is stored and having the decryption key held by select authorized users keeps data safe even in the event of a data breach. All data stores and servers should be encrypted at all times until decrypted by authorized user access. |
| Encryption at flight | Encryption in flight is the process of encrypting and protecting data that is being sent over a network or connection that is not inside the server. By encrypting the information, we send out to our clients or workers we can ensure that any leaked information is not immediately useful to those who would want to steal it. All data sent out by software developed by our company must be encrypted where authorized users are given a decryption key to be used when receiving information from our servers. |
| Encryption in use | Encryption in use is the process of keeping all data encrypted even as it is accessed by authorized users. Data that is being access by a user is decrypted per request and encrypted afterwards to ensure no data is stored in a plain text form. This ensures data leaked through different hacking techniques remains useless no matter when they were able to pull the information from the servers. Information created or received by software provided by our company must have encryptions set in place to encrypt data before and after authorized user access. No data used by our software must be stored in any non-encrypted form. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is where we pair users with a unique ID and password that will serve to prove their identity when attempting to access information or functions of our software. By pairing users to accounts we can keep access of company data limited to those with accounts and further limit data access by their permission level. All company data must only be accessible by users with authorized accounts. New user accounts must have identities verified before permissions are given. |
| Authorization | Authorization is where we limit the data that accounts can access by giving them levels of permission. This ensures that no user can access all of the data on the server and prevents users with less permissions from accessing more sensitive data. All user accounts created must be given a designation level at the time of creation to determine data permissions. Only accounts designated as administrator and IT should be able to make changes to existing databases. Base level users should only be able to read accessible files once they are sent to the server database. |
| Accounting | Accounting is the recording of resource use by users. This can include who accessed data, how long they accessed it, and what changes were made. Logs that are created for this purpose can then be used for audits and security checks should data corruption occur. This can be especially useful for tracking down the last user to access a corrupted piece of data for questioning. Data logs should be in place to record all resources accessed by our software. Data logs should take note of what user accessed each piece of data. |

**\***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 | 01/22/2023 | Standard creation | Jacob McPherson | [Insert text.] |
| 3.0 | 02/12/2023 | Risk assessment and Policy creation | Jacob McPherson | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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