**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 | Validating all untrusted data sources is necessary to secure against vulnerabilities. BY doing so we can eliminate bad data that can lead to various problems. These sources include multiple external locations. That may include but not limited to: CLI arguments, network interfaces, environment variables, and user-controlled files. Validating data from these sources ensures that we are not taking data that seeks to do harm by exploiting vulnerabilities that may exist and in doing so ensures the security of the application. |
| 1. Heed Compiler Warnings | When building the application and code, a variety of warnings may show up in your environment but will not prevent the application from building or running. While easy to ignore these, they should not be. These warnings should be read carefully and we should make changes to remove the warnings. We should also use the latest and highest level tools for warnings available in our compiler. This ensures we can catch all the warnings and address them to prevent security issues from arising because of them. |
| 1. Architect and Design for Security Policies | The best practice is to create an architecture before designing. Once the architecture is created then you should seek to design the software and implement security practices. This can be controlling access levels by use of subdivision. Creating a system where the architecture is multiple pieces working towards the whole ensures only those who need access to specific parts have them. This mitigates security risk by ensuring unauthorized users are unable to interact with pieces they have no business in. |
| 1. Keep It Simple | The more complex a design is, the more room for error will manifest. Working in smaller and simpler designs ensures that such errors can be minimalized. By working smaller we can also ensure the security of the program and code a lot easier since there are less moving parts to worry about. |
| 1. Default Deny | Build security policies based on permission access. The default action for security should be to deny access. The policy should read the conditions and permit access based upon it. If the conditions are met then and only then will access be permitted otherwise it will deny. |
| 1. Adhere to the Principle of Least Privilege | Build policies based around minimizing privilege levels to perform tasks. Avoid running as an elevated user. This ensures that privileges are granted to run what is necessary and that alone. This ensures that elevated access privileges are not unnecessarily granted and prevents malicious elevated tasks from running such as code injections or execution of arbitrary code. |
| 1. Sanitize Data Sent to Other Systems | When dealing with complex subsystems it becomes necessary to sanitize or destroy the data. Proper sanitization of the data ensures the recoverability of it can not be utilized by external malicious sources for use in various injection based attacks. This should occur after the call but before invocation of the subsystem. |
| 1. Practice Defense in Depth | Utilize layered security and multiple security components. This ensures that when one of your security protocols fails to prevent an attack another one can immediately step in and mitigate the attack. This process ensures that the security is thick enough that to perform a valid and damaging attack, the malicious actor needs to be actively and deliberately attempting to breach the system. It also allows for more chance of deterrence when they are met with multiple security layers and allow you to answer to the attack in a more timely fashion with less of the probability of exploitation occurring. |
| 1. Use Effective Quality Assurance Techniques | Using good and multiple Quality Assurance techniques helps in the elimination of vulnerabilities. By utilizing these we can check for vulnerabilities and weaknesses we may not see during the coding phases. It also allows the system to be viewed by different perspectives and how they can identify vulnerabilities. |
| 1. Adopt a Secure Coding Standard | To properly code securely, you should analyze your programming language and platform and develop on standard. While there are best practices, a standardized rule set is not defined in the industry. You (and your team) should develop a secure coding standard to be utilized and adhered to. This ensures you are following a secure coding protocol as well as having it be identified easily and use by all involved in the project. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Do not apply operators expecting one type to data of an incompatible type** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Weak typing in C allows type casting memory to different types. Because the internal representation of most types is system dependent, applying operations intended for data of one type to data of a different type will likely yield non-portable code and produce unexpected results. |

| **Noncompliant Code** |
| --- |
| This example attempts to increment an integer that’s set as a float. This can lead to unexpected and unwanted behavior. |
| float f = 0.0;  int i = 0;  float \*fp;  int \*ip;    assert(sizeof(int) == sizeof(float));  ip = (int\*) &f;  fp = (float\*) &i;  printf("int is %d, float is %f\n", i, f);  (\*ip)++;  (\*fp)++;    printf("int is %d, float is %f\n", i, f); |

| **Compliant Code** |
| --- |
| This example shows that the pointers are set to the compatible data types. As such the behavior should run as expected. |
| float f = 0.0;  int i = 0;  float \*fp;  int \*ip;    ip = &i;  fp = &f;  printf("int is %d, float is %f\n", i, f);  (\*ip)++;  (\*fp)++;    printf("int is %d, float is %f\n", i, f); |

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

| **Principles(s):** 10 Adopt a Secure Coding Standard & 2 Adhere to Compiler Warnings. This standard discusses how its possible to cast the wrong data type to a field and result in unexpected results. In most likelihoods the compiler may output a warning about this issue that could be ignored, and the program would still run. By adhering to the warning, we can ensure that the issue is dealt with before it becomes an issue. This is also a result of weak type casting and can be dealt with by adopting a coding standard and adhering to it. Doing so ensures that developers are following a standard set of rules that prevents these types of issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2022.1 | PORTING.BITFIELDS  PORTING.CAST.FLTPNT  PORTING.CAST.PTR  PORTING.CAST.PTR.FLTPNT  PORTING.CAST.PTR.SIZE  PORTING.CAST.SIZE | Static Code Analysis tool that uses SEI Cert standards. Can check for casting and pointers for incorrect data type casting. |
| LDRA tool suite | 9.7.1 | 554 S | Code Analysis and Quality tool suite that incorporates coding standards to check for casting. |
| PRQA QA-C++ | 4.4 | 3017, 3030 | Defect Prevention and Analysis tool that incorporates multiple standards. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ std::basic\_string template requires no such character. |

| **Noncompliant Code** |
| --- |
| Because char buf[12] is limited to 12 characters, this can lead to a buffer overflow. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Using std::string is the better choice as it does not bind to a character limitation as in the preceding example. This helps guard against the likelihood of buffer overflows. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** 1: Validate Input Data. This principle is about ensuring your data is good and matches for its use as to prevent errors. Since this standard is ensuring the data will fit where its going, it should fall into this principle. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No Space for Null Terminator  Buffer Overrun  Type Overrun |
| Parasoft C/C++ Test | 2021.2 | CERT\_SPP-STR50-b  CERT\_SPP-STR50-c  CERT\_SPP-STR50-e  CERT\_SPP-STR50-f  CERT\_SPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflow from tainted Data  Avoid buffer write overflow from tainted data  Do not use ‘char’ buffer to store input from ‘std::cin’ |
| Polyspace Bug Finder | R2021b | CERT C++:STR50-CPP | Checks for:  Use of dangerous standard function  Missing null in string array  Buffer overflow from incorrect string format specifier  Destination buffer overflow in string manipulation |
| Helix QAC | 2022.1 | C++ 2835, C++ 2836, C++ 2839, C++ 5216 | Static code analysis tool that adheres to SEI Cert standards |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Range check element access** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The std::string index operators const\_reference operator[](size\_type) const and reference operator[](size\_type) return the character stored at the specified position, pos. When pos >= size(), a reference to an object of type charT with value charT() is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| This attempts to replace the first character in a string with a capital letter. When the string is empty, this results in undefined behavior. |
| #include <string>  #include <locale>    void capitalize(std::string &s) {  std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

| **Compliant Code** |
| --- |
| In this version, a check to see if the string is empty occurs. If it is, then the character replacement does not run preventing the undefined behavior. |
| #include <string>  #include <locale>    void capitalize(std::string &s) {  if (s.empty()) {  return;  }    std::locale loc;  s.front() = std::use\_facet<std::ctype<char>>(loc).toupper(s.front());  } |

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

| **Principles(s):** 1:Validate Input Data & 9 Use Effective Quality Assurance Techniques. This standard seems to adhere to both these principles. The concept is to securely code for when an out of range exception occurs. This can mean validating the input data and whether it exists. If it does not exist a call to an indiex in what we expect will result in errors. Quality assurance would mean we are testing and building to ensure that the data sent and the range we call for are both in existence as well. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Helix QAC | 2022.1 | C++3162, C++3163, C++3164, C++3165 | Static code analysis tool adhering to SEI Cert standards |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-STR53-a | Guarantee that container indices are within valid range |
| Polyspace Bug Finder | R2021b | CERT C++:STR53-CPP | Checks for:   * Array access out of bounds * Array Access with tainted index * Pointer dereference with tainted offset |

#### 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** |
| --- |
| In this, a second argument can be passed in that would enable the ability to return information regarding the system. Thus a malicious actor can obtain elevated access to the system. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| This utilizes whitelisting. It allows for a check on the characters entered and does not permit the bad characters and sanitizes them to prevent the ability to pass secondary arguments. |
| 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):** 7 Sanitize Code Sent to Other Systems & 9 Use Effective Quality Assurance Techniques. This standard is about sanitizing strings so that any undesirable text in strings that can be used for attacks is eliminated. Principle 7 discusses a form of this with sanitizing after sending so the string is not recoverable and able to be used for an attack. I see this fits because of the core idea of sanitizing the string before and after transmission is the most effective manner to secure it. Principle 9 discusses using checks for vulnerabilities, since this standard is about coding to remove the possibility of injections it would definitely qualify as a mean s of quality assurance to remove the injection text from the string. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command Injection  Format String Injection  LDAP Injection  Library Injection  SQL Injection  Untrusted Library Load  Untrusted Process Creation |
| Parasoft C/C++ test | 2021.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |
| Polyspace Bug Finder | R2021a | CERT C:Rec. STR02-C | Checks For:   * Execution of externally controlled command * Command executed from externally controlled path * Library Loaded from externally controlled path |
| Coverity | 6.5 | TAINTED\_STRING | Static code analysis tool to check strings |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Allocate sufficient memory for an object** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | The types of integer expressions used as size arguments to malloc(), calloc(), realloc(), or aligned\_alloc() must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer. |

| **Noncompliant Code** |
| --- |
|  |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,  int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(tmb));  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm) {  .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,  .tm\_mday = day, .tm\_mon = mon, .tm\_year = year  };  return tmb;  } |

| **Compliant Code** |
| --- |
| Best practice is to use the size of operator. It does not evaluate the operand and handles unreferenced and null pointer. |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,  int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(\*tmb));  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm) {  .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,  .tm\_mday = day, .tm\_mon = mon, .tm\_year = year  };  return tmb;  } |

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

| **Principles(s):** 10 Adopt a Secure Coding Standard. This standard discusses size arguments for memory allocations to prevent overflows. The code still will run in theory if this standard is not used. However, adopting it is a more secure and defined manner of doing so. Adopting this as a standard fits within the definitions of principle 10 as it discusses adopting a defined standard of secure coding practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  ALLOC.SIZE.TRUNC  IO.TAINT.SIZE  MISC.MEM.SIZE.BAD | Addition overflow of allocation size  Addition overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Truncation of allocation size  Tainted allocation size  Unreasonable size argument |
| Astrée | 20.10 | Malloc-size-insufficient | Checks for direct rules violations and undefined behavior from invalid memory access |
| PolySpace Bug Fnder | R2021a | CERT C: Rule MEM35-C | Checks for:   * Pointer access out of bounds * Memory allocation with tainted size |
| Parasoft C/C++ test | 2021.2 | CERT\_C-MEM35-a | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via 'malloc', 'calloc' or 'realloc' function |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Do not abruptly terminate the program** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | The std::abort(), std::quick\_exit(), and std::\_Exit() functions are used to terminate the program in an immediate fashion. They do so without calling exit handlers registered with std::atexit() and without executing destructors for objects with automatic, thread, or static storage duration. How a system manages open streams when a program ends is implementation-defined [ISO/IEC 9899:1999]. Open streams with unwritten buffered data may or may not be flushed, open streams may or may not be closed, and temporary files may or may not be removed. Because these functions can leave external resources, such as files and network communications, in an indeterminate state, they should be called explicitly only in direct response to a critical error in the application. |

| **Noncompliant Code** |
| --- |
| Calling f() may result in a termination call. This is due to throw exception. |
| #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** |
| --- |
| Enabling the throwing\_func ensures that f() handles all exceptions. This prevents further throws from occurring. |
| #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):** 7 Sanitize Data Sent to Other Systems & 10 Adopt a Secure Coding Standard. This standard addresses what happens when an abort or exit type call occurs. In which the program can leave open streams when called to these actions. This fits with 7 as it discusses how to properly close streams and thus sanitize any data that was opened before moving on. This also falls in line with 10 as it is about adopting this standard and defining it in the security policy to be adhered to. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of Abort  Use of Exit |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-ERR50-a  CERT\_CPP-ERR50-b  CERT\_CPP-ERR50-c  CERT\_CPP-ERR50-d  CERT\_CPP-ERR50-e  CERT\_CPP-ERR50-f  CERT\_CPP-ERR50-g  CERT\_CPP-ERR50-h  CERT\_CPP-ERR50-i  CERT\_CPP-ERR50-j  CERT\_CPP-ERR50-k  CERT\_CPP-ERR50-l  CERT\_CPP-ERR50-m  CERT\_CPP-ERR50-n | The execution of a function registered with 'std::atexit()' or 'std::at\_quick\_exit()' should not exit via an exception  Never allow an exception to be thrown from a destructor, deallocation, and swap  Do not throw from within destructor  There should be at least one exception handler to catch all otherwise unhandled exceptions  An empty throw (throw;) shall only be used in the compound-statement of a catch handler  Exceptions shall be raised only after start-up and before termination of the program  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point  Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)  Function called in global or namespace scope shall not throw unhandled exceptions  Always catch exceptions  Properly define exit handlers  The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  Avoid throwing exceptions from functions that are declared not to throw  The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |
| PolySpace Bug Finder | R2021b | CERT C++:ERR50-CPP | Checks for implicit call to terminate() function |
| Helix Qac | 2022.1 | C++5014 | Static code analysis tool to check for use of operations against SEI Standards |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. |

| **Noncompliant Code** |
| --- |
| Neither function is setup to handle the thrown exception. This can lead to an abrupt termination call to terminate(). |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| The entry point is set to handle all exceptions. This ensures proper handling of the exception throws. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** 9 Use Effective Quality Assurance Techniques. This standard discusses that all exceptions must be handled. If they are not, then it can leave to early terminations which can leave data open and accessible. This falls in with 9 as good quality assurance will check to ensure that vulnerabilities don’t exist as well as that the product is in optimal working order. Since unhandled exceptions can lead to attacks and does not optimize the product this seems a direct connection to policy 9. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catches exceptions  Each exception explicitly thrown in code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2021b | CERT C++:ERR51-CPP | Checks for unhandled exceptions |
| RuleChecker | 20.10 | Main-function-catch-all  Early-catch-all | Catches all exceptions |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ - ERR51 | Checks against ERR51 standards for exception handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Copy operations must not mutate the source object** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Copy operations (copy constructors and copy assignment operators) are expected to copy the salient properties of a source object into the destination object, with the resulting object being a "copy" of the original. What is considered to be a salient property of the type is type-dependent, but for types that expose comparison or equality operators, includes any properties used for those comparison operations. This expectation leads to assumptions in code that a copy operation results in a destination object with a value representation that is equivalent to the source object value representation. Violation of this basic assumption can lead to unexpected behavior. |

| **Noncompliant Code** |
| --- |
| This example copies the data to a and changes the original m to 0. All copy functions afterwards appear as 0. This not the intended or expected result. |
| #include <algorithm>  #include <vector>    class A {  mutable int m;    public:  A() : m(0) {}  explicit A(int m) : m(m) {}    A(const A &other) : m(other.m) {  other.m = 0;  }    A& operator=(const A &other) {  if (&other != this) {  m = other.m;  other.m = 0;  }  return \*this;  }  int get\_m() const { return m; }  };  void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

| **Compliant Code** |
| --- |
| This example does not change the original source of m. This ensures that it operates as intended for all future references to it. |
| #include <algorithm>  #include <vector>    class A {  int m;    public:  A() : m(0) {}  explicit A(int m) : m(m) {}    A(const A &other) : m(other.m) {}  A(A &&other) : m(other.m) { other.m = 0; }    A& operator=(const A &other) {  if (&other != this) {  m = other.m;  }  return \*this;  }    A& operator=(A &&other) {  m = other.m;  other.m = 0;  return \*this;  }    int get\_m() const { return m; }  };    void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

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

| **Principles(s):** 4 Keep it Simple. This standard discusses that when you perform a copy operation the original data should remain in tact as to prevent undesired results or issues with it. This can be violated by attempt steps for an operation that are too complex and unnecessary for what you are trying to accomplish. Principle 4 can be utilized here to advise you to keep it simple and this will help keep you from making these kinds of errors that come with overly complex designs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-OOP58-a | Checks for copy operations mutate source object |
| Polyspace Bug Finder | R2021b | CERT C++: OOP58-CPP | Checks for copy operation modifying source operand |
| PRQA QA-C++ | 4.4 | 4075 | Static code analysis tool that adheres to SEI Cert and checks for modified operand at source |
| Klocwork | 2022.1 | CERT.OOP.COPY\_MUTATES | Static Code analysis tool that checks for mutated source during copy |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input/Output | [STD-009-CPP] | A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first. |

| **Noncompliant Code** |
| --- |
| This code opens the file and terminates. The file is left open and never closed thus allowing its resources to remain open and accessible. |
| #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 code opens the file and closes it later on before terminating. This ensures the file resources are not accessible when the program terminates. |
| #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):** 7 Sanitize Data Sent to Other Systems, 9 Use Effective Quality Assurance Techniques. This standard discusses when you read a file in that you should close it before function termination occurs. This ensures that the file is not left open and can not be accessed through illicit methods or by other aspects of the program. This in effect a way of sanitizing data by removing access to information in temporary memory spaces. This also ensures the quality of the program by properly handling all in and outs and dealing with them in the proper terms of opening and closing as would be in effect a form of quality assurance and preventative measure to ensure unwanted access to a file is not feasible. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | ALLOC.LEAK | Checks for data open and considers it a leak |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-FIO51-a | Checks resources and ensures they are freed |
| Polyspace Bug Finder | R2021b | CERT C++: FIO51-CPP | Checks for resource leaks |
| Helix QAC | 2022.1 | C++4786,C++4787,C++4788 | Static Code Analysis tool that checks against SEI Cert standards. Will Check for memory leak and Close(). |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Avoid deadlock by locking in a predefined order** |
| --- | --- | --- |
| Concurrency | [STD-010-CPP] | Mutexes are used to prevent multiple threads from causing a data race by accessing the same shared resource at the same time. Sometimes, when locking mutexes, multiple threads hold each other's lock, and the program consequently deadlocks. Four conditions are required for deadlock to occur:   * mutual exclusion (At least one non-shareable resource must be held.), * hold and wait (A thread must hold a resource while awaiting availability of another resource.), * no preemption (Resources cannot be taken away from a thread while they are in-use.), and * circular wait (A thread must await a resource held by another thread which is, in turn, awaiting a resource held by the first thread.).   Deadlock needs all four conditions, so preventing deadlock requires preventing any one of the four conditions. One simple solution is to lock the mutexes in a predefined order, which prevents circular wait. |

| **Noncompliant Code** |
| --- |
| This code is susceptible to deadlock when thr1 locks the mutex for ba2 while thr2 locks the mutex for ba1. |
| #include <mutex>  #include <thread>    class BankAccount {  int balance;  public:  std::mutex balanceMutex;  BankAccount() = delete;  explicit BankAccount(int initialAmount) : balance(initialAmount) {}  int get\_balance() const { return balance; }  void set\_balance(int amount) { balance = amount; }  };    int deposit(BankAccount \*from, BankAccount \*to, int amount) {  std::lock\_guard<std::mutex> from\_lock(from->balanceMutex);    // Not enough balance to transfer.  if (from->get\_balance() < amount) {  return -1; // Indicate error  }  std::lock\_guard<std::mutex> to\_lock(to->balanceMutex);    from->set\_balance(from->get\_balance() - amount);  to->set\_balance(to->get\_balance() + amount);    return 0;  }    void f(BankAccount \*ba1, BankAccount \*ba2) {  // Perform the deposits.  std::thread thr1(deposit, ba1, ba2, 100);  std::thread thr2(deposit, ba2, ba1, 100);  thr1.join();  thr2.join();  } |

| **Compliant Code** |
| --- |
| This version enables manual ordering. As such a defined order of when instances occur in deposit. As such, this helps prevent the deadlock by ensuring tasks happen in a specific order. |
| #include <atomic>  #include <mutex>  #include <thread>    class BankAccount {  static std::atomic<unsigned int> globalId;  const unsigned int id;  int balance;  public:  std::mutex balanceMutex;  BankAccount() = delete;  explicit BankAccount(int initialAmount) : id(globalId++), balance(initialAmount) {}  unsigned int get\_id() const { return id; }  int get\_balance() const { return balance; }  void set\_balance(int amount) { balance = amount; }  };    std::atomic<unsigned int> BankAccount::globalId(1);    int deposit(BankAccount \*from, BankAccount \*to, int amount) {  std::mutex \*first;  std::mutex \*second;    if (from->get\_id() == to->get\_id()) {  return -1; // Indicate error  }    // Ensure proper ordering for locking.  if (from->get\_id() < to->get\_id()) {  first = &from->balanceMutex;  second = &to->balanceMutex;  } else {  first = &to->balanceMutex;  second = &from->balanceMutex;  }  std::lock\_guard<std::mutex> firstLock(\*first);  std::lock\_guard<std::mutex> secondLock(\*second);    // Check for enough balance to transfer.  if (from->get\_balance() >= amount) {  from->set\_balance(from->get\_balance() - amount);  to->set\_balance(to->get\_balance() + amount);  return 0;  }  return -1;  }    void f(BankAccount \*ba1, BankAccount \*ba2) {  // Perform the deposits.  std::thread thr1(deposit, ba1, ba2, 100);  std::thread thr2(deposit, ba2, ba1, 100);  thr1.join();  thr2.join();  } |

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

| **Principles(s):** 3 Architect and Design for Security Policies, 9 Use Effective Quality Assurance Techniques, 10 Adopt a Secure Coding Standard. This standard discusses the symptoms needed for a system to cause a deadlock and how to properly code to prevent it. By preventing deadlock, we can code it properly so the only the correct systems can access the resources necessary and only in a specific order we can define to ensure it falls in line with Principle 3. By doing these we are also preventing vulnerabilities we would not normally find as the bad code would not flag any warnings or errors to us and aligning with Principle 9. Since there is possibility that the incorrect code can run fine and only certain circumstances may cause the deadlock, it becomes best to avoid the possibility altogether and would fall in line with principle 10 which would be a proactive means to addressing such issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | CONCURRENCY.LOCK.ORDER | Checks for Conflicting Lock Order |
| PolySpace Bug Finder | R2021b | CERT C++:CON53-CPP | Checks for deadlock |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-CON53-a | Do not acquire locks in different order |
| PRQA QA-C++ | 4.4 | 1772,1773 | Enforces MTA Roles, Static Code Analysis tool, checks for deadlock and orders |

### 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.

Automations can occur at multiple locations and time throughout the DevOps process. Much of which may be based on the feature being worked on. During pre-production automation can be performed during the build or verify and test segments. This should be dependent on the tools and what they are looking for. Some tools can be integrated into a development environment and be used to identify security issues as you build the program. Verify and Test staging is where most automations for pre-production would run. At this point the program should be actively run through analysis and automation tools to check for security issues.

During the production phase, automations can be run in all sections. While each section performs a different aspect of the testing, they all work towards identifying, testing, and remediating security issues. Automations can be used in any of these phases to help identify what security vulnerabilities exist, offer insight on how to correct the issues, and check again once patched to see if new vulnerabilities now exist.

### 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 | Medium | Probable | Medium | P8 | L2 |
| STD-002-CPP | High | Likely | Medium | P18 | L1 |
| STD-003-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Probable | High | P6 | L2 |
| STD-006-CPP | Low | Probable | Medium | P4 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Likely | Low | P9 | L2 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Low | Probable | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This is the process of protecting data that is not actively in use by way of algorithms generally. This policy applies as it is a means of protecting data that’s not being used. It is one method of protecting things like customer sensitive data or to help prevent ransomware attacks. |
| Encryption at flight | This is the process of protecting data that is in transit from one location to another. This may occur in the form of encrypted packets or secured transit methods. Since the modern world of computing is web driven, development needs to corporate such methods to protect data as it travels from location to another. This can help protect the data and prevent attacks. |
| Encryption in use | This is the process of protecting data that is active and often placed in volatile storage mediums. By encrypting this data, it is important for developers to protect the data thats been allocated to memory and prevent it from illicit activities to access it. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This process is part of security framework of identifying who is trying to access a system. The policy applies as it is necessary to identify who is attempting to access a program in order to verify whether that person should have access to that information. This is necessary to safeguard data that can be accessed through the system. |
| Authorization | This process dictates what the user can see and determines the security policies that are active for that user. By defining security policies to specific roles and assign users to those roles we can control what information the user can see. This becomes important to prevent users who have no need to access specific information can not access it. This ensures the integrity of the data and prevent unauthorized modifications to data. |
| Accounting | This is the process of logging and tracking all events and actions taken. This ensures that there is a trackable document that shows who did what and when to ensure the accountability of a system, program, or data. This becomes important as it is necessary to track changes and events so when something goes wrong, the user can be tracked back to for purposes of identifying what was done so it can be undone or for legal reasons if the actions were malicious. |

**\***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 | 04/08/2022 | Updated To include standards, risks, summary, automations | Chris Gollnick |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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