**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 | All inputs from data sources not trusted should be validated. This prevents malicious users from taking advantage of vulnerabilities in the software. Data sources such as user-controlled files, command line arguments, and network interfaces should not be trusted easily. Being wary of external data sources such as these will help create more secure code. |
| 1. Heed Compiler Warnings | To find the most vulnerabilities, code should be compiled to display as many warnings as possible. Once warnings are identified, the code should be altered to resolve them. To find any remaining flaws in the code both dynamic and static analysis tools can be used. Once found, eliminate the flaws to prevent them from being vulnerabilities. |
| 1. Architect and Design for Security Policies | Security policies should be considered throughout the architecting and design process of your software. The software should not only actively implement security policies, but also enforce them. By considering how to best integrate security policies into the system one can design a program which is fundamentally more secure. |
| 1. Keep It Simple | Secure code is simple and DRY (do not repeat yourself). By keeping the code simple there is less places where vulnerabilities can occur. Complex, sprawling code can make it difficult to find vulnerabilities. Vulnerabilities that are not identified can then pop up when the software is used. Part of keeping code simple is also keeping it DRY, preventing unnecessary repetition in code also makes it more secure. This is because if logic is repeated and a change is made to one part but not the other holes in security could then be made. |
| 1. Default Deny | As a default all access should be denied. This is because there is far less instances where access should be permitted then denied. Rather than defaulting to permit access except in certain circumstances, everyone except a select few should be denied access. If as a default access was permitted then malicious users could easily penetrate security and cause damage. |
| 1. Adhere to the Principle of Least Privilege | No job should be executed with more privileges than necessary. If a higher level of privileges is granted then it must be applied for a minimum time. This way, it is more difficult for a malicious user to take advantage of privileges and do what they want with the system. |
| 1. Sanitize Data Sent to Other Systems | Before passing data to other systems, data should be sanitized by the calling process. Malicious users can use an injection attack to invoke unused functionality and pass their own code to other systems. By sanitizing the data before passing it along the possibility of this occurring is decreased. Commercial off-the-shelf components, command shells, and relational databases are all things that should be sanitized by the calling process. |
| 1. Practice Defense in Depth | Defense in depth is a strategy which seeks to improve security through the use of layers. By identifying where your layers are vulnerable and purposefully stacking them in such a way that they vulnerabilities are covered, a more secure system can be created. The thought behind this strategy is that even if one layer is penetrated, there are still other layers left protecting the system. This is more secure than attempting to create one very strong layer of protection. |
| 1. Use Effective Quality Assurance Techniques | Do research to find the best quality assurance techniques for whatever system is being worked on. They should not only detect, but also destroy vulnerabilities in the system. There are many different kinds of tests out there but source code audits, fuzz testing, and penetration testing should all be included during the quality assurance phase. External reviewers who are independent from the system can be beneficial as they give a non-biased review. They may find some issues that were overlooked due to assumptions or simply being missed. |
| 1. Adopt a Secure Coding Standard | Secure coding standards that are set in place should be followed. Developers should also keep in mind their own circumstances so that they can adopt the best standards for their situation. Circumstances such as the language being used should be considered while choosing coding standards. If necessary, coding standards can be developed for the specific system being worked on. |

### 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-001-CPP] | Do not cast to an out-of-range value  This standard is set in place to prevent underflow or overflow from occurring. Developers should be aware of the range of their values so they can stay within them. Staying in scope will prevent underflow or overflow. |

| **Noncompliant Code** |
| --- |
| The code below attempts to cast a total to charTotal that is outside the signed character range of +127. Because of this, buffer overflow will occur resulting in incorrect behavior by the program. |
| #include <assert.h>  #include <iostream>  int main()  {  signed char charTotal, char1;  char1 = 100;  charTotal = char1 \* 2;  } |

| **Compliant Code** |
| --- |
| We have fixed the code so that the expression will not result in a value outside of the signed characters range. This range is -128 to 127. Please note that different variables have different ranges so this is not a solution that will work in every situation. |
| #include <assert.h>  #include <iostream>  int main()  {  signed char charTotal, char1;  char1 = 100;  if ((char1 \* 2) < -128 || (char1 \* 2) > 127)  {  // Handle error  }  else {  charTotal = char1 \* 2;  }  } |

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

| **Principles(s):** 2, heed compiler warnings; 9, use effective quality assurance techniques; 10, adopt a secure coding standard  When developers choose which data type to use they need to be aware of its range. If they are casting to an out of range value they should be notified by the compiler and should take heed of this. There should also be effective quality assurance techniques in place to catch when a data value is out of range. Having a secure coding standard will help ensure developers chooser the correct data type each time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | cast-integer-to-enum | Partially checked |
| CodeSonar | 7.1p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Parasoft C/ C++test | 2022.1 | CERT\_CPP-INT50-a | All enum expressions will contain only values that correspond with enumerators of the enumeration |
| RuleChecker | 22.10 | cast-integer-to-enum | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Values should be referenced using only valid iterators, references, or pointers  When referencing an element in a container only valid iterators, references, and pointers should be used. It is easy for containers to be modified, invalidating the iterators, references, and pointers stored within them. Developers should keep this in mind so nothing invalid is used to reference elements in a container. |

| **Noncompliant Code** |
| --- |
| When an insertion occurs at the beginning or end of the double ended queue (deque) all iterators are invalidated while the references stay valid. Note that inserting in the middle of the deque results in all references and iterators becoming invalidated. References and pointers *can* be independently invalidated for the same operation; however, this is not secure. In this situation the pointer may stay valid even when the reference is invalidated, but it may point to a different data value than expected. Avoid doing this when coding. |
| #include <algorithm>  #include <deque>  #include <iterator>    void shoppingCart(const double \*products, std::size\_t numProducts) {  std::deque<double> d;  auto position = d.begin();  for (st::size\_t i = 0; i < numProducts; ++i; ++position)  {  d.insert(position, products[i] + 41.0);  }  } |

| **Compliant Code** |
| --- |
| Transform is a generic standard template library algorithm. Elements are accepted by it so they can be transformed and then stored in a specified location. Here, the location is at the beginning of the indexed sequence container “d”. Because it is inserted at the beginning of deque, all element references stay valid. |
| #include <algorithm>  #include <deque>  #include <iterator>    void shoppingCart(const double \*products, std::size\_t numProducts) {  std::deque<double> d;  std::transform(products, products + numProducts, std::inserter(d,  d.begin()), [](double d) { return d + 41.0; });  } |

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

| **Principles(s):** 2, heed compiler warnings; 9, use effective quality assurance techniques; 10, adopt a secure coding standard  Values should be referenced using only valid iterators, references, or pointers. If an invalid one is used then the compiler should issue a warning and a change should be made. Using effective quality assurance will help catch invalid references as well. Adopting a secure coding standard will also ensure that developers do not do this. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.UAF | Use after free |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-CTR51-a | While iterating over a container it should not be modified |
| Polyspace Bug Finder | R2022b | CERT C++: CTR51-CPP | Partial coverage over invalid iterator usage |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Strings will not be created using any of the following examples:  -From a null pointer  -Without using valid references, iterators, and pointers to reference basic\_string elements  -Without proper space for data  -Without range checking element access  Doing any of the above things will result in an error with your string. By avoiding them you can ensure string correctness. |

| **Noncompliant Code** |
| --- |
| Here, our code is noncompliant because read() is being used with inputBuffer. This array is unformatted and does not contain a null terminator. Because of this, when the string constructor is called using inputBuffer undefined behavior occurs. This results in an incorrect string. |
| #include <fstream>  #include <string>    void readInput(std::istream &input) {  char inputBuffer[20];  try {  input.read(inputBuffer, sizeof(inputBuffer));  } catch (std::ios\_base::failure &error) {  // Catch error & correctly respond  }    std::string myString(inputBuffer);  // ...  } |

| **Compliant Code** |
| --- |
| The above issue of creating an incorrect string is solved by constructing a string the size of the input, with this method a null terminator is not inserted and rather a correct string object is created. Please note this solution only works when the size of the input is known. |
| #include <fstream>  #include <string>    void readInput(std::istream &input) {  char inputBuffer[20];  try {  input.read(inputBuffer, sizeof(inputBuffer));  } catch (std::ios\_base::failure &error) {  // Catch error & correctly respond  }  std::string myString(inputBuffer, input.gcount());  // ...  } |

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

| **Principles(s):** 2, heed compiler warnings; 9, use effective quality assurance techniques; 10, adopt a secure coding standard  To ensure string correctness developers should pay attention to warnings issued by the compiler. Effective quality assurance techniques will also help identify incorrect strings, and a secure coding standard will ensure all strings are created correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.MEM.NPD | Null pointer dereference |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2022b | CERT C++: STR51-CPP | Full coverage checking for string operations on null pointers |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-STR52a | Elements of a basic\_string should use valid iterators, references, and pointers |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C] | When passing data to complex sub systems all data should be sanitized.  As data is passed, certain elements within it may cause undesired actions or commands to be triggered. These are considered software vulnerabilities and can be taken advantage of by malicious users to attack via SQL Injection. |

| **Noncompliant Code** |
| --- |
| Here is an example from Sun Solaris TELNET daemon, vulnerability [VU#881872] (“STR02-C. Sanitize data passed to complex subsystems”, 2022). This noncompliant code contains a flaw which allows malicious users to log in to a system as an authorized user. In the code below, unsanitized data is passed into execl() as an argument and can successfully gain access to the system. |
| (void) execl(LOGIN\_PROGRAM, "login",  "-p",  "-d", slavename,  "-h", host,  "-s", pam\_svc\_name,  (AuthenticatingUser != NULL ? AuthenticatingUser :  getenv("USER")),  0); |

| **Compliant Code** |
| --- |
| We have fixed the issue from above by adding “—” before the getenv(“USER”) line. Inserting the double dash signals getopt() (a function in POSIX used by the login program) to halt. After the double dash no argument passed into the login program is interpreted therefore solving the SQL Injection attack program. |
| (void) execl(LOGIN\_PROGRAM, "login",  "-p",  "-d", slavename,  "-h", host,  "-s", pam\_svc\_name,  "--",  (AuthenticatingUser != NULL ? AuthenticatingUser :  getenv("USER")), 0); |

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

| **Principles(s):** 1, validate input; 3, architect and design for security policies; 7, Sanitize data sent to other systems; 9, use effective quality assurance techniques.  To prevent SQL Injection all data should be sanitized, this helps prevent software vulnerabilities. Quality assurance should be used to catch vulnerabilities that can be taken advantage of by SQL Injection, and the program should be architected and designed in a way that prevents this. All input should also be validated if it is from an untrusted source to also help eliminate vulnerabilities in the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | IO.INJ.SQL | SQL injection |
| Parasoft C/C++test | 2022.1 | CERT\_C-STR02-c | Protect against SQL Injection |
| Polyspace Bug Finder | R2022b | CERT C: Rec. STR02-C | Partial coverage.  Commands and libraries from paths that are externally controlled are checked.  Checks the execution of commands that are externally controlled. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | All dynamically allocated resources are to be properly deallocated  To protect your memory, you should properly deallocate it. To expand upon this, make sure you are properly allocating your memory as well. If incorrectly deallocated functions are pointed to then undefined behavior can occur, this is a vulnerability. |

| **Noncompliant Code** |
| --- |
| Here, the pointer “structure” is deleted before any memory was returned after using new Structure . This results in undefined behavior, something needs to happen to the memory before it can be deleted. |
| #include <iostream>    struct Structure {  Structure() { std::cout << "Structure::Structure()" << std::endl; }  ~Structure() { std::cout << " Structure::~Structure()" << std::endl; }  };    void doStuff() {  alignas(struct Structure) char space[sizeof(struct Structure)];  Structure \*structure = new Structure;    // ...    delete structure;  } |

| **Compliant Code** |
| --- |
| We solve the problem from above by using the destructor of structure (~Structure). It is no longer neccessary to use delete(). |
| #include <iostream>    struct Structure {  Structure() { std::cout << "Structure::Structure()" << std::endl; }  ~Structure() { std::cout << " Structure::~Structure()" << std::endl; }  };    void doStuff() {  alignas(struct Structure) char space[sizeof(struct Structure)];  Structure \*structure = new Structure;    // ...    structure->~Structure;  } |

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

| **Principles(s):** 2, **heed compiler warnings; 9, use effective quality assurance techniques; 10, adopt a secure coding standard.**  All dynamically allocated resources are to be properly deallocated, if this is not done the compiler should give a warning identifying areas that need to be fixed. Effective quality assurance will also help catch any resources that are not properly deallocated. By adopting a secure coding standard developers can also avoid doing this. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Clang tidy will check for this, however, not all variations of the rule are caught with this |
| CodeSonar | 7.1p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| Polyspace Bug Finder | R2022b | CERT C++: MEM51-CPP | Partial coverage, checks for invalid freeing and deletion of pointers, and deallocation of pointers which have been previously deallocated |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Use assertions to incorporate diagnostic testing into your program  Assertions are a great tool to test that the assumptions made during programming are correct. While helpful, they can affect the speed of your build. Because of this, assertions are not always kept in the code when it is deployed. As a rule of thumb, assertions should not be used in such a way that it would affect your code when removed. Please note that keeping assertions in your code after deployment can help troubleshoot your code when unexpected bugs occur. Therefore, it is not always necessary to remove them. |

| **Noncompliant Code** |
| --- |
| In the code below we set the string equal to “Hello World!”. The assertion is checking for the string to equal that, however, at some point during coding the string was actually changed to “Hello Moon!”. Because this was forgotten about assert fails and an error is printed. |
| #include <stdio.h>  #include <assert.h>    int main()  {  string myString = “Hello World!”;    // . . .  myString = “Hello Moon!”;    assert(myString == “Hello World!”);    // . . .    return 0;  } |

| **Compliant Code** |
| --- |
| In this very simple example, it is asserted that the size of charLength is less than or equal to the max length of char divided by the size of char \*. If this fails, an error is thrown. If it does not then charSentence can be set to charLength \* sizeof(char \*). |
| #include <assert.h>  assert(charLength <= SIZE\_MAX/sizeof(char \*));  charSentence = charLength \* sizeof(char \*); |

**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  Assertions can be used to incorporate diagnostic testing into your program. The Architecting and designing of assertions should be well thought out so they improve security. Proper quality assurance techniques will ensure the diagnostic tests are effective. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.FUNCS.ASSERTS | Check if you have enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Detects for side effects in operation/function calls |
| Parasoft C/C++test | 2022.1 | CERT\_C-MSC11-a | Internal invariants and assumptions are documented through the use of a liberal amount of assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions  All exceptions should be handled to avoid undefined behavior. If an exception is not handled correctly, the process may abnormally terminate. Denial-of-service (DoS) attacks are most likely to happen when a process is terminated abnormally. To avoid this, make sure to catch all thrown exceptions with a matching exception handler. |

| **Noncompliant Code** |
| --- |
| In this example, the program will unexpectedly terminate. This is because we have a function throwing an error, however, there is no exception handler. The error is thrown and without anything to catch it the program terminates. You must *always* handle your exceptions to avoid this from occurring. |
| void functionWithError() noException(false);    void exceptionThrower() {  functionWithError();  }    int main() {  exceptionThrower();  } |

| **Compliant Code** |
| --- |
| We use a try…catch statement to handle the error. This catches the error and then executes the necessary code to handle it. This prevents the program from terminating, protecting it from a DoS attack. |
| void functionWithError() noException(false);    void exceptionThrower() {  try {  exceptionThrower ();  } catch (…) {  // Logic for handling an exception  }  } |

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

| **Principles(s):** 2, heed compiler warnings; 3, architect and design for security policies; 9, use effective quality assurance techniques.  All exceptions should be handled. An exception that is not handled will cause the compiler to issue a warning, this should be heeded. When architecting and designing your program you should consider security, therefore exceptions should be designed in such a way that improves security. Quality assurance should catch if an exception is not handled correctly so that it can be fixed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partial checking |
| CodeSonar | 7.1p0 | LANG.STRUCT.UCTCH | Looks for unreachable catches |
| LDRA tool suite | 9.7.1 | 527 S | Partial implementation |
| Polyspace Bug Finder | R2022b | CERT C++: ERR51-CPP | Partial coverage of unhandled exception checking |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory | [STD-008-CPP] | A pointer that is already owned is not to be stored in an unrelated smart pointer  Pointer values that construct smart pointers are owned by that smart pointer. You can relinquish ownership through modifiers like release, and reset, however, it is important to understand that reset will result in the pointer value being destroyed. Also note that some smart pointers can share underlying pointers, these are called with shared\_ptr(). This does not make multiple smart pointers own the same pointer, however. In this case, only the first smart pointer owns the original pointer value, and the rest are simply “related”. |

| **Noncompliant Code** |
| --- |
| Here, we have a pointer “myPointer”. The code then immediately tries to create two smart shared pointers using the same value “myPointer”. While shared pointers *can* share pointer values, only one can be the true owner. Because both smart pointers were constructed from the same pointer, vulnerabilities are exposed when the smart pointers are deleted. |
| #include <memory>    void function() {  int \*myPointer = new int;  std::shared\_ptr<int> smartPointer1(myPointer);  std::shared\_ptr<int> smartPointer2(myPointer);  } |

| **Compliant Code** |
| --- |
| In this example the vulnerabilities from before are eliminated. This is done by creating a shared smart pointer and then initializing the second one with the first smart pointer. By doing this, if smartPointer2 is deleted the source pointer (in this case smartPointer1) is not deleted. If smartPointer1 is deleted, then because there are no more shared pointers sharing the value the memory is destroyed. This prevents dangling pointers and vulnerabilities. |
| #include <memory>    void function() {  std::shared\_ptr<int> smartPointer1 = std::make\_shared<int>();  std::shared\_ptr<int> smartPointer2(smartPointer1);  } |

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

| **Principles(s):** 2, heed compiler warnings; 9, use effective quality assurance techniques; 10, adopt a secure coding standard.  A pointer that is already owned is not to be stored an unrelated smart pointer, the compiler should notify you if this is done. If the compiler fails to notify you then quality assurance techniques should catch the error. Adopting a secure coding standard will help prevent this from being done in the first place. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-MEM56-a | Unrelated smart pointers are not to be used to store a already-owned pointer value |
| Polyspace Bug Finder | R2022b | CERT C++: MEM56-CPP | Full coverage of checking for pointers that are already owned |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations | [STD-009-CPP] | Declarations are to be syntactically clear  Never write syntactically ambiguous declarations. Sometimes, syntax can be interpreted as a declaration or as an expression statement. When this happens, the compiler tries to determine the result, but it is not guaranteed to be correct. Avoid ambiguity to ensure no incorrect behavior occurs. |

| **Noncompliant Code** |
| --- |
| In this example the program never actually prints its statement. The reason that it cannot print is because of ambiguous code. When myWidget() is declared, Widget is supposed to execute the print statement. It does not because the compiler cannot tell if myWidget() is a function pointer declaration with no arguments, or if Widget is being declared as a local variable. This confusion makes the print statement fail to execute. |
| #include <iostream>    struct Widget {  Widget() { std::cout << "Widget Constructed" << std::endl; }  };    void function() {  Widget myWidget();  } |

| **Compliant Code** |
| --- |
| Ambiguity is removed by replacing the parentheses with curly braces. This is known as a “braced-init-list”, and it will allow the statement to print by initializing a local variable. The other option is to omit the parentheses, this declares a variable and not a function. A function without parameters is not created this time, but rather a local variable. This allows the function to be correctly called and the print statement executes. |
| #include <iostream>    struct Widget {  Widget() { std::cout << "Widget Constructed" << std::endl; }  };    void function() {  // Acceptable  Widget myWidget{};  // OR can also do this  Widget myWidget1;    } |

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

| **Principles(s):** 10, adopt a secure coding standard; 9, use effective quality assurance techniques  Declarations are to be syntactically clear, adopting a secure coding standard will help ensure this is true. Effective quality assurance techniques should catch any syntactically ambiguous statements so that they can be fixed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.STRUCT.DECL.FNEST | Nested function declaration |
| LDRA tool suite | 9.7.1 | 296 S | Partial implementation |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b  CERT\_CPP-DCL53-c | No parentheses around function declaration parameter names, and variable declaration local variable names  Avoid function declarations that are syntactically ambiguous |
| Polyspace Bug Finder | R2022b | CERT C++: DCL53-CPP | Checks for confusion between function parameter or unnamed object declaration, and object and function declaration. Full coverage |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-010-CPP] | Valid ordering predicate is to be used at all times  Infinite loops or erratic behavior can occur when invalid ordering predicate is used for an associative container. This also occurs when the sorting algorithm is used with comparison criterion that is ordered invalidly. |

| **Noncompliant Code** |
| --- |
| Here, we have an object using a comparator incorrectly. It does not stick to the strict weak ordering requirement causing the program to fail to return the correct value. It returns true when it should instead be returning false when values are equivalent. After iterating, unspecified behavior occurs which causes vulnerabilities. |
| #include <functional>  #include <iostream>  #include <set>    void ordering() {  std::set<int, std::less\_equal<int>> example{4, 12, 24};  for (auto exRange = example.equal\_range(10); exRange.first !=  exRange.second; ++exRange.first) {  std::cout << \*exRange.first << std::endl;  }  } |

| **Compliant Code** |
| --- |
| The invalid comparator std::less\_equal is removed and the default comparator is used. This makes the code return correct responses and avoid unspecified behavior. |
| #include <iostream>  #include <set>    void ordering() {  std::set<int> example{4, 12, 24};  for (auto exRange = example.equal\_range(10); exRange.first !=  exRange.second; ++exRange.first) {  std::cout << \*exRange.first << std::endl;  }  } |

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

| **Principles(s):** 10, adopt a secure coding standard  Valid ordering predicate is to be used at all times, adopting a secure coding standard will help ensure that this rule is adhered too. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-CTR57-a | Comaparison functions that return true for equal values should never be used with associative containers |
| Polyspace Bug Finder | R2022b | CERT C++: CTR57-CPP | Partial coverage of strict weak ordering |

### 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.

Automation is a great tool for enforcing standards and improving security. Verify and test is a great place to start for automation. It is becoming increasingly common to have automated tests, especially for redundant processes. Automating tests for redundant processes can reduce human error as you do not have to manually repeat the same thing multiple times which increases the likelihood of making a mistake. It can also save money as you do not need to pay a human to do it. Tools can be used to scan for vulnerabilities, including those created from invalid data types, incorrect strings, and ambiguous syntax. By using automated tests you can increase your programs security before release.

During deployment it is also possible to use automation. You can use it to issue a health check, ensuring everything was deployed correctly. This is crucial as it is imperative that you notice any issues as soon as the system is deployed. You can also you it for monitoring and detecting issues post deployment. Mechanisms should be in place to check for malicious attacks, security breaches, and any discovered vulnerabilities. The system should monitor when the tools it is using have new versions available. By doing this, updates can be made as quickly as possible, reducing the time malicious users have to take advantage of vulnerabilities. When responding to attacks automation can also be used. The system should automatically block and turn off services to prevent further damage from occurring.

### 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 | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | High | Probably | High | P6 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-C | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-C | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| STD-009-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-010-CPP | Low | Probable | High | P2 | 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 | Encryption in rest refers to protecting data even when it is not being used. Data at rest is that which is not used on a daily basis. Although data is often protected behind a firewall, encryption needs to be used to improve security. To encrypt this data a transformation occurs. The data is taken out of its original form and encrypted into a different form that can not be understood by outsiders. Only those with authorized access are able to transform the data back into its original form. |
| Encryption at flight | Data in flight refers to data that is transferred over a network, it is moving from one place to the next. Data is very vulnerable during this process and so it is vital for it to be properly protected. Encryption at flight refers to encrypting this data in flight so it is protected during transit. During this process raw data is encrypted, transferred, and then unencrypted when it is ready to be accessed. |
| Encryption in use | With encryption in use data is secure at every lifecycle stage. When data becomes unencrypted, access is controlled limiting threats to it. This is very beneficial as it protects data at all times, limiting the opportunities for malicious users to access that data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is properly identifying users while preventing malicious ones from posing as someone they are not. Good authentication requires uses to enter unique information that identifies them, this is user login. This information also needs to be protected in order to prevent malicious users from pulling out internal credentials and taking advantage of them. |
| Authorization | Authorization refers to who can issue certain commands. Companies need to have secure authorization processes in place to prevent malicious users from issuing commands they should not be able to. Authorization handles who can make changes to the database, who can add new users, and the level of access a user can reach. This is very important as you do not want a malicious user to create a new user, give it a high level of access, and then make changes to the database. |
| Accounting | Accounting means tracking what resources are used by a user while accessing the system. Accounting should track how much data is being set in a session, where it is being sent to, how long the user has had a session for, how often they do certain things. Keeping track of this data will make it a lot easier to find a data breach as you can easily track which files were accessed by users as well as other processes. |

**\***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] | 12/4/2022 | Project One | Kathryn McNeil | [Insert text.] |
| [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 |

Resources

Brightline IT. (2017, May 17). *What is encryption at rest, and why is it important for your business?* https://brightlineit.com/encryption-at-rest-important-business/

Carnegie Mellon University. (2020, May 29). *SEI CERT C++ coding standard.* [*https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682*](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682)

Carter, E. (2021, July 7). *The role of automation in software development lifecycle*. Kaholo. https://kaholo.io/blog/the-role-of-automation-in-software-development-lifecycle/

Cppreference.com (n.d.) *C++ reference.* [*https://en.cppreference.com/w/*](https://en.cppreference.com/w/)

Cplusplus.com. (n.d.). *Reference: Standard C++ library reference*. <https://cplusplus.com/reference/>.

Das, P. (n.d.). *In-use encruption- What it is and how companies benefit.* Sotero. https://www.soterosoft.com/blog/data-in-use-encryption-data-in-motion-encryption/#:~:text=In%2DUse%20encryption%20takes%20a,%2C%20cloud%2C%20or%20hybrid).

Faso, V. L. (2015, April 28). *How to protect data in flight.* Skillsoft Global Knowledge. https://www.globalknowledge.com/us-en/resources/resource-library/articles/how-to-protect-data-in-flight/

GeeksforGeeks. (2022, February 28). *Assertions in C/C++*. https://www.geeksforgeeks.org/assertions-cc/#:~:text=Assertions%20are%20statements%20used%20to,void%20assert(%20int%20expression%20)%3B

Jacob Sorber. (2017, August 2). *Find bugs faster using assertions* [video]. Youtube. <https://www.youtube.com/watch?v=1Jh9BUxIw0U>.

Robert Seacord. (2022, April 27). *STR02-C. Sanitize data passed to complex subsystems.* Carnegie Mellon University Software Engineering Institute. <https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems>

Seacord, R. (2013). *Secure Coding in C and C++*. Addison-Wesley.

Seacord, R. (2011, March 1). *Top 10 secure coding practices.* Secure Coding. <http://web.archive.org/web/20160404152049/https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices>.

TechTarget Contributor. (n.d.). Authentication, authorization, and accounting (AAA). TechTarget. https://www.techtarget.com/searchsecurity/definition/authentication-authorization-and-accounting#:~:text=Authentication%2C%20authorization%2C%20and%20accounting%20(AAA)%20is%20a%20term,necessary%20to%20bill%20for%20services.