**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 inputs from untrustworthy data sources can help prevent many of the most serious software flaws. External data sources, such as command line arguments, environmental variables, and network interfaces, should always be considered a source of risk. |
| 1. Heed Compiler Warnings | When building your code, always utilize the highest warning levels allowed and remove warnings by altering the code. Use static and dynamic analysis techniques to assist find and remove additional security issues. |
| 1. Architect and Design for Security Policies | Make sure you adopt and enforce the appropriate security regulations while creating your software architecture. |
| 1. Keep It Simple | To avoid having to adapt to complicated systems that are more subject to faults and failed security methods, keep your code design short and straightforward. |
| 1. Default Deny | Rather than excluding people, basing your access decisions on permission. Access should be prohibited by default, with criteria allowing access. |
| 1. Adhere to the Principle of Least Privilege | Process executions should have the fewest privileges possible to perform the task. Elevated permissions should only be used for the duration of the work at hand. This is to help prevent attackers from running arbitrary code with those increased privileges. |
| 1. Sanitize Data Sent to Other Systems | Data passing through complicated subsystems should be sanitized. To deter attackers from manipulating subsystem components via injection attacks. |
| 1. Practice Defense in Depth | Always use numerous levels of protection to train defense in depth. So, if one layer fails, another is in place to keep security defects from becoming exploitable vulnerabilities. |
| 1. Use Effective Quality Assurance Techniques | Techniques for quality assurance can improve the odds of detecting and removing flaws. Multiple testing steps, independent security assessments, and external security reviews can all help to ensure that systems are safer. |
| 1. Adopt a Secure Coding Standard | Always use a secure coding standard for development, regardless of the language or platform. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not cast to an enumeration value that is out of range. |

| **Noncompliant Code** |
| --- |
| Checks if a provided value is inside the permitted enumeration value range. It's possible that the type won't be able to represent the specified integer value once it's been cast. |
| enum EnumType {  First,  Second,  Third  };  void f (int intVar) {  EnumType enumVar = static\_cast(intVar);  if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| Before executing the conversion, the conforming solution verifies the value represented by the enumeration type to ensure that the conversion does not result in an undefined value. As a result, the converted value is limited to a single enumerator type. |
| enum EnumType {  First,  Second,  Third  };  void f (int intVar) {    if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast(intVar);  } |

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

| **Principles(s):** It's conceivable that unspecified values will cause a buffer overflow, allowing an attacker to run arbitrary code. Enumerators are infrequently used for indexing into arrays or other kinds of pointer arithmetic; therefore this scenario is more likely to result in data integrity issues than uncontrolled code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | function-ellipsis | Fully checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL50 |  |
| Clang | 3.9 | cert-dcl50-cpp | Checked by clang-tidy. |
| CodeSonar | 5.4p0 | LANG.STRUCT.ELLIPSIS | Ellipsis |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | To refer to elements of a container, use valid references, pointers, and iterators. |

| **Noncompliant Code** |
| --- |
| After the first call to insert () in this noncompliant code sample, pos is invalidated, and further loop iterations have unknown behavior. |
| #include <deque>  void f (const double \*items, std: size\_t count) {  std::deque d;  auto pos = d.begin();  for (std::size\_t i = 0; i < count; ++i, ++pos) {  d.insert (pos, items[i] + 41.0);  }  } |

| **Compliant Code** |
| --- |
| On each insertion, pos is given a valid iterator in this conforming approach, preventing undefined behavior. |
| #include <deque>  void f (const double \*items, std::size\_t count) {  std::deque d;  auto pos = d.begin();  for (std::size\_t i = 0; i < count; ++i, ++pos) {  pos = d. insert (pos, items[i] + 41.0);  }  } |

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

| **Principles(s):** Undefined behavior occurs when incorrect references, pointers, or iterators are used to reference items of a container. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | reserved-identifier |  |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL51 | Partially checked |
| clang | 3.9 | -Wreserved-id-macro  -Wuser-defined-literals | The -Wreserved-id-macro switch is not enabled by default or when using -Wall, but it is when using -Weverything. This flag does not detect all instances of this rule, such as when reserved names are renamed. |
| PVS – Studio | 7.07 | V783 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Don't try to make a std::string out of a null reference. |

| **Noncompliant Code** |
| --- |
| A std::string object is formed from the results of a call to std::getenv in this noncompliant code sample ().  However, because std::getenv() fails with a null pointer, this code might result in undefined behavior if the environment variable isn't there (or some other error occurs). |
| #include <cstdlib>  #include <string>  void f () {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| The results of the call to std::getenv() are tested for null before the std::string object is formed in this compatible approach. |
| #include <cstdlib>  #include <string>  void f () {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code (Jack 2007, van Sprundel 2006). The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| low | unlikely | low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL52 | [Insert text.] |
| Parasoft C/C+ +test | 2021.1 | CERT\_CPP-DCL52-a | Never use the terms 'const' or 'volatile' to describe a reference type. |
| Polyspace Bug finder | R2020A | CERT\_CPP-DCL52-a | Const-qualified reference types are checked.  Changes to constqualified reference types  The rule is completely covered. |
| PRQA QA-C++ | 4.4 | 0014 | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not save the value of an already-owned smart pointer in an unrelated smart pointer. |

| **Noncompliant Code** |
| --- |
| Two unrelated smart pointers are formed from the same underlying pointer value in this noncompliant code sample. The pointer value it handles is deleted when the local, automated variable p2 is removed.  The local, automated variable p1 then deletes the identical pointer value when it is destroyed, leading in a double-free weakness. |
| #include  void f () {  int \*i = new int;  std::shared\_ptr p1(i);  std::shared\_ptr p2(i);  } |

| **Compliant Code** |
| --- |
| The std::shared ptr objects are associated to one another in this compatible manner through copy construction.  The use count for the shared pointer value is decremented but not zero when the local, automated variable p2 is removed. The usage count for the shared pointer value is then decremented to zero, and the managed pointer is removed when the local, automated variable p1 is erased. Instead of creating a raw reference and storing its value in a local variable, this compatible alternative uses std::make shared(). |
| #include <memory>  void f () {  std::shared\_ptr p1 = std::make\_shared<int>();  std::shared\_ptr p2(p1); } |

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

| **Principles(s):** Passing a point object to a deallocation method that hasn't been received by the matching allocation function results in undefined behavior, which can lead to exploitable flaws. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| low | likely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 296 S | Partially implemented |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-DCL53-a CERT\_CPP-DCL53-b | Always define functions in the scope of a file.  An identifier defined in a local or function prototype scope cannot be used to obscure an identifier declared in a global or namespace scope. |
| Polyspace Bug Finder | R2020A | CERT\_C+=:DCL53-CPP | Needs to check for statements that might be mistaken with each other, such as function and object declarations.  Declaration of an unnamed object or function argument  The rule is completely covered. |
| PRQA QA-C++ | 4.4 | 2502, 2510 | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Deallocate dynamically distributed resources correctly. |

| **Noncompliant Code** |
| --- |
| The local variable space is provided as the expression to the placement new operator in this noncompliant code sample. The pointer produced by that function is subsequently handed to::operator delete(), which causes undefined behavior because::operator delete() is attempting to free memory that was not returned by::operator new (). |
| #include <iostream>  struct S {S () {std::cout << "S::S()" << std::endl; }  ~S () {std::cout << "S::~S()" << std::endl; }  };  void f () {alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;  // ...  Delete s1:  } |

| **Compliant Code** |
| --- |
| Instead of calling::operator delete(), this compatible approach directly calls s1's destructor. This is one of the rare instances where directly executing a destructor is necessary. |
| #include <iostream>  Struct S {S () {std::cout << "S::S()" << std::endl; }  ~S () {std::cout << "S::~S()" << std::endl; }  };  void f () {alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...  s1->~S();  } |

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

| **Principles(s):** Sending a point object to a deallocation method that hasn't been received by the matching allocation function results in undefined behavior, which can lead to exploitable flaws. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| low | Likely | low | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | new-delete-pairwise | Partially checked |
| Clang | 3.9 | misc-new-delete-overloads | Clang-tidy was used to make sure everything was in order. |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL54-a | Always add new and remove old at the same time. |
| Polyspace Bug finder | R2020a | CERT C++ CPP: DCL54-CPP | Needs to check for a conflict between both the overloaded new and remove operators (rule completely covered). |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-cpp] | To check the value of a constant expression, use a static assertion. |

| **Noncompliant Code** |
| --- |
| The assert() macro is used in this noncompliant code to assert a property of a memory-mapped structure that is required for the code to work correctly: |
| #include <assert.h>  struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned  int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| A preprocessor conditional statement, such as this compatible solution, can be used for assertions containing just constant expressions: |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned  int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** Static assertion is a useful diagnostic tool for identifying and removing software flaws that could lead to vulnerabilities during compilation. The lack of static assertions, on the other hand, does not imply that the code is erroneous. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion  Bauhaus Suite | 6.9.0 | CertC++-DCL55 |  |
| Parasoft c/c++test | 2020.2 | CERT\_CPP-DCL55-a | It's not a good idea to send a reference to a structure to a function that can transfer data to user space. |
| CodeSonar | 6.0p0 | (customization) | Users can create a custom check that reports assert() macro usage. |
| Compass/Rose |  | [Insert text.] | Could discover breaches of this rule by searching for assert() calls, and if it can evaluate the assertion (due to all values being known at compilation time), then static-assert should be used instead; this implies ROSE can recognize macro invocation. |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Before main() starts operating, handle any exceptions that are thrown. |

| **Noncompliant Code** |
| --- |
| The constructor for S in this noncompliant example may throw an exception that is not caught when globalS is built at program startup. |
| struct S {  S() noexcept(false);  };  static S globalS; |

| **Compliant Code** |
| --- |
| This compliant solution converts globalS into a local variable with a static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for S is called the first time the function globalS() is called rather than at program startup, allowing any exceptions thrown during object construction to be caught. This solution does need modifying source code to replace past usage of globalS with a function call to globalS. (). |
| struct S { S() noexcept(false);  };  S &globalS() {  try {  static S s;  return s;  } catch (...) { //  Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable  } |

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

| **Principles(s):** Throwing an uncaught exception causes irregular program termination and can result in denial-of-service attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | potentially-throwing- static-initialization | Partially Checked |
| Axivion Bauhaus Suite | 6.9.0 | CERTC++-ERR58 | [Insert text.] |
| clang | 3.9 | Cert-eer58-cpp | Checked by clang-tidy |
| Helix QAC | 2021.1 | [Insert text.] | Exception should only be raised only after application has started and before it has finished. |
| Parasoft C/C++ test | 2020.2 CERT\_CPP- ERR58-a |  |  |
| PRQA QA-C++ | 4.4 | 4634, 4636, 4637, 4639 |  |
| Rule Checker | 20.10 | potentially-throwing-static- initialization | Partially checked |

#### 

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input output | [STD-008-CPP] | Without an intermediate positioning call, do not alternately input and output from a file stream. |

| **Noncompliant Code** |
| --- |
| This example of noncompliant code appends data to the end of a file before reading from the same file.  The behavior is unclear since there is no intervening positioning call between the formatted output and input calls. |
| #include  #include    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  file << "Output some data";  std::string str;  file >> str;  } |

| **Compliant Code** |
| --- |
| The std::basic istreamT>::seekg() function is called between the output and input in this conforming approach, which eliminates the undefined behavior. |
| #include  #include void f(const std::string &fileName)  { std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error return;  }  file << "Output some data";  std::string str;  file.seekg(0, std::ios::beg);  file >> str;  } |

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

| **Principles(s):** [It's unclear behavior to alternately input and output from a stream without a flush or positioning call in between. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | destructor-without-noexcept  delete-without-noexcept | Fully checked |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP\_FIO50-a | Without an intermediary, alternately inputting and outputting from a stream. Do not switch between input and output from a stream without a flush or positioning call in between. |
| Polyspace Bug Finder | R2020a | ECRT C++: FIO50-CPP | Needs to check for alternating input and output from a stream without flushing or calling for positioning (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programing | [STD-009-CPP] | Constructors and destructors should not call virtual functions (). |

| **Noncompliant Code** |
| --- |
| The base class attempts to capture and release an object's resources by calling virtual methods from the constructor and destructor in this noncompliant code sample. The B::B() constructor, on the other hand, uses B::seize() instead of D::seize() (). Similarly, the B::B() destructor uses B::release() instead of D::release() (). |
| struct B {  B() { seize(); }  virtual ~B() { release(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  B::seize();  // Get derived resources...  }  void release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| The base class attempts to capture and release an object's resources by calling virtual methods from the constructor and destructor in this noncompliant code sample. The B::B() constructor, on the other hand, uses B::seize() instead of D::seize() (). Similarly, the B::B() destructor uses B::release() instead of D::release() (). |
| Class b {  void seize\_mine();  void release\_mine();  public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }  protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); } };  class D : public B {  void seize\_mine();  void release\_mine();  public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }  protected: void seize() override {  B::seize();  seize\_mine(); }  void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):** Designing and building and designing safety policies - writing code to avoid flaws  Keep things simple - this rule should always be followed, as keeping code as minimal as possible is great practice.  Make tests that are as effective as feasible using excellent quality assurance techniques.  Adopt a secure coding standard - prioritizing security helps to avoid vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | unlikely | medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | virtual-call-in-constructor invalid\_function\_pointer | Fully checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-OOP50 | [Insert text.] |
| clang | 3.9 | clang-analyzer- alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Helix QAC | 2021.1 | [Insert text.] | [Insert text.] |
| LDRA tool suite | 9.7.1 | 467S, 92D | Fully implemented |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-OOP50-a  CERT\_CPP-OOP50-b  CERT\_CPP-OOP50-c  CERT\_CPP-OOP50-d | Constructors should not call virtual functions.  Destructors should not call virtual functions.  Do not use any of the class's constructors to call its virtual functions.  Do not call the virtual functions of a class from its destructor. |
| PVS-Studio |  | Virtual-call-in-customer | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | S1699 |  |
|  |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | [STD-010-CPP] | All exit pathways must return a value for value returning functions. |

| **Noncompliant Code** |
| --- |
| Because the programmer omitted to return the input value for positive input in this noncompliant code sample, not all code pathways yield a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| All code paths in this conforming solution now yield a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

| **Principles(s):** When a value-returning function fails to return a value from a code path, undefined behavior occurs, which could be exploited to cause data integrity violations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wreturn-type | This rule isn't caught in all cases, such as function-try-blocks. |
| CodeSonar | 6.0p0 | LANG.STRUCT.MRS | A return statement is missing. |
| Helix QAC | 2021.1 | [Insert text.] | [Insert text.] |
| LDRA tool suite | 9.7.1 | 2 D, 36 S | Completely implemented |
| Parasoft C/C++ tes | 2020.1 | CERT\_CPP-MSC52-a | An explicit return statement with an expression must be present on all exit paths from a function having a non-void return type. |
| Polyspace bug finder | R2020a | Cert C++: MSC52-a | Involves checking for any return Finder statements that are missing (rule partially covered) |
| SonarQube C/C++ plugin | 4.10 | S935 |  |
| PRQA QA- C++ | 4.4 | 1510 |  |
| PVS-Stuido | 7.07 | V591 |  |
| RuleChecker | 20.10 | Return-implicit | Checked thoroughly |

### 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.

[Insert your written explanations here.]

### 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 | High | Unlikely | Medium | P4 | L3 |
| STD-001-CPP | High | Probable | Medium | P12 | L1 |
| STD-002-CPP | Low | Unlikely | low | P3 | L3 |
| STD-003-CPP | Low | Unlikely | low | P3 | L3 |
| STD-004-CPP | Low | Unlikely | medium | P2 | L3 |
| STD-005-CPP | Low | Probable | low | P6 | L2 |
| STD-006-CPP | Low | Unlikely | high | P1 | L3 |
| STD-007-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-008-CPP | Low | Unlikely | Medium | P6 | L2 |
| STD-009-CPP | high | Likely | Medium | P6 | L2 |
| STD-010-CPP | Medium | Unlikely | 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 | Encryption at rest is a technique for preventing an attacker from accessing unencrypted data while it is stored on disk. If an attacker acquires a hard disk containing encrypted data but not the encryption keys, he or she must decrypt the data in order to read it. |
| Encryption at flight | The process of encrypting data as it is being sent over the internet. Data may be unencrypted when at rest on disk arrays in some applications, such as remote replication, but encrypted while being sent to ensure protection. |
| Encryption in use | Access to encrypted data at rest and in motion is made possible by compromising data in use.  Someone who has access to random access memory, for example, can parse it to find the encryption key for data in transit. They can decode encrypted data in the wild once they've got the encryption key. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Compromise of data in use allows access to encrypted data at rest and in motion.  Anyone with access to random access memory, for example, can parse it to discover the encryption key for data in transit. Once they have the encryption key, they can decode encrypted data in the wild. |
| Authorization | The level of access a user has within the system is defined by authorization. This might include whether the user has permission to read, create, remove, or edit files in the database. This can also lead to a user's ability to add or remove files and users from the system. |
| Accounting | Accounting is the method of keeping track of what a user does with their degree of system access. This will record which databases are visited, what was done when they were accessed, and who first accessed the system. |

**\***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

4- a straightforward answer that isn't overly complex

5- By standard, all accounts are denied access until they are granted access.

6- keep privileges from sneaking up on you.

8- it's a part of the whole, but it's not the entire

10- it is a necessary aspect of the procedure

* Firewall logs

4- a straightforward answer that isn't overly complex.

5- By nature, all accounts are denied access until they are granted access.

6- keep privileges from sneaking up on you

7- avoid the transmission of unneeded data

8- it's a part of the whole, but it's not the entire

10- it is a necessary aspect of the procedure

* Anti-malware logs

4- a straightforward answer that isn't overly complex.

5- By default, all accounts are denied access until they are granted access.

6- keep privileges from sneaking up on you.

7- avoid the transmission of unneeded data.

8- it's a part of the whole, but it's not the entire.

10- it is a necessary aspect of the procedure

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 |  | First Revision |  | [Insert text.] |
| 1.2 |  | Final Revision |  | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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