**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 input data to a program, user or not, should be sanitized and searched through for any unwanted input into a program that might cause unwanted results due to code. Unintended results can cause vulnerabilities within a program that may be exploited. Look out for warnings line command arguments if possible. |
| 1. Heed Compiler Warnings | Compiler warnings are important for a developer to pay attention to. Compiler warnings could give hints that data is acting out of bounds with an unintended result. Use different tools to identify any issues encountered. Compile and test constantly to catch errors. |
| 1. Architect and Design for Security Policies | Good Architecture and design provides integrity, confidentiality, and protection against deliberate attacks against data and systems. Being aware of the security policies in place for the architecture in question is important. |
| 1. Keep It Simple | Code should be straightforward and structed in such a way as to not confuse the programmer or others. Use correct naming conventions and indentation, and proper security processes. . |
| 1. Default Deny | Access should be defined by the Admin and users should only be able to access per levels of permission. It prevents anyone without proper access from getting unintended results from the system. |
| 1. Adhere to the Principle of Least Privilege | Principle of least privilege is when a user account is only given permissions that are essential to perform for its intended use. This works hand in hand with Default deny to help a program determine which user should get which permissions. |
| 1. Sanitize Data Sent to Other Systems | Any and all data sent to other systems from this program needs to be checked and ensure that the data is free of any possible vulnerabilities to prevent exploits to a system. |
| 1. Practice Defense in Depth | Defense in depth gives systems a lot of protection. Any layer can be breached given time, but multiple layers can help protect a system that would be breached if only one defense was used. |
| 1. Use Effective Quality Assurance Techniques | Proper QA is a vital part of every program and process. Having dedicated time and personnel to check over code for issues and vulnerabilities is important to ensure that there are no flaws. |
| 1. Adopt a Secure Coding Standard | A Secure Coding standard is imperative to have when developing for a program or a platform. Coding with these policies and standards in mind can lead to more efficient and safer code. |

### 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 define a C-style variadic function |

| **Noncompliant Code** |
| --- |
| This function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior |
| #include <cstdarg>    int add(int first, int second, ...) {  int r = first + second;  va\_list va;  va\_start(va, second);  while (int v = va\_arg(va, int)) {  r += v;  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| A variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. |
| #include <type\_traits>    template <typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Arg s) { return f + s; }    template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Ts... rest) {  return f + add(rest...);  } |

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

| * + - 1. Validate input data- searching through input data into a variable argument is important so a program does not have unintended side effects.       2. 2. Heed Compiler warnings- multiple tools will throw warning for this issue and should not be ignored. Refer to tools sections that use static code analyzers.       3. Keep it simple- variable functions should be simple and have a set number of arguments. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL50 | Static Code analysis tool for next generation analysis using MISRA. |
| Astree | 20.10 | Function-elipsis | Static Code analysis tool for runtime errors. |
| Clang | 3.9 | Cert=dcl50-cpp | Objective Compiler |
| CodeSonar | 7.1.p0 | LANG.STRUCT.ELLIPSIS | Static Analysis tool for SAST solutions |

The tools for this standard will be straightforward. Static code checkers will check for C-style variadic functions in the code and will display additional compiler warnings. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not cast to an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| This code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| 1. Validate input data- searching through input data and not casting to an out of range data value is very important so a program does not have unintended side effects. 2. Architect and Design for security Policies- it is important to know the security policies. 3. Sanitize data sent to other systems- casting to an out of range value and sending the result out can have unintended consequences. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1.p0 | LANG.CAST.COERCE / LANG.CAST.VALUE | Static Analysis tool for SAST solutions |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | Static Code analysis tool for next generation analysis using MISRA. |
| Parasoft | 2022.1 | CERT\_CPP-INT50-a | Test Automation |
| PRQA QA-C++ | 4.4 | 3013 | Commercial Static Analysis tool |

The tools for this standard will be straightforward. Static code checkers will check for out of range enumerations in the code and will display additional compiler warnings. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std:: string from a null pointer |

| **Noncompliant Code** |
| --- |
| For this example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #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.**

| 1. Validate input data- creating a string from a null pointer can cause issues with your program. 2. Architect and Design for security Policies- it is important to know the security policies, a string from a null pointer can be a security threat. 3. Sanitize data sent to other systems- creating a string from a null pointer and sending the result out can have unintended consequences. 4. Use effective QA techniques- effective QA can help prevent issues like this from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Assert\_failure | Static Code analysis tool for runtime errors. |
| CodeSonar | 7.1.p0 | LANG.MEM.NPD | Static Analysis tool for SAST solutions |
| Klocwork | 2022.3 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT | Static Code analysis tool |
| Parasoft | 2022.1 | CERT\_CPP-STR51-a | Test Automation |

The tools for this standard will be straightforward. Static code checkers will check for strings created from null pointers in the code and will display additional compiler warnings. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 4

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

| **Noncompliant Code** |
| --- |
| Two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability. |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| The std::shared\_ptr objects are related to one another through copy construction. When the local, automatic variable p2 is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local variable. |
| #include <memory>    void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| 1.Validate input data- Do not store an already-owned pointer value in an unrelated smart pointer, it can cause a double vulnerability.  2.Architect and Design for security Policies- it is important to know the security policies, a string from a null pointer can be a security threat. A double vulnerability can be exploited.  3.Sanitize data sent to other systems- creating a string from a null pointer and sending the result out can have unintended consequences, and is a security issue.  4.Use effective QA techniques- effective QA can help prevent issues like this from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft | 2022.1 | CERT\_CPP-MEM56-a | Test Automation |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM56 | Static Code analysis tool for next generation analysis using MISRA. |
| Astree | 20.10 | dangling\_pointer\_use | Static Code analysis tool for runtime errors. |
| PVS-Studio | 7.21 | V1006 | Test Automation |

The tools for this standard will be straightforward. Static code checkers will check for an already-owned pointer value in an unrelated smart pointer. in the code and will display additional compiler warnings. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Detect and handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

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

| 1.Architect and Design for security Policies- it is important to know the security policies, and a memory allocation error can cause many problems if not properly handled.  2. Practice defense in depth- proper memory allocation can help improve the defense in depth.  3.Use effective QA techniques- effective QA can help prevent issues like this from occurring, and with memory allocation being very important should have a lot of attention paid to ensure this does not happen. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.21 | V522  V668 | Test Automation |
| Klocwork | 2022.3 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.CALL.MUST  NPD.FUNC.MIGHT | Static Code analysis tool |
| Coverity | V7.5.0 | CHECKED\_RETURN | Static Analysis |
| Helix QAC | 2022.3 | C++ 3225  C++ 3226  C++ 3227  C++ 3228  C++3229 | Static Analysis |

The tools for this standard will be straightforward. Static code checkers will check potential unhandled memory allocation errors in the code and will display additional compiler warnings. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave 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** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| 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.**

| 1. Validate Input Data- All input data to a program, user or not, should be sanitized and searched  through for any unwanted input into a program that might cause unwanted results due to code.  2. Keep It Simple- Code should be straightforward and structed in such a way as to not confuse the programmer or others. Use correct naming conventions and indentation, and proper security processes. Assert tests if something is true and should not be used to test things that are not going to return true or false.  3.Use effective QA techniques- effective QA can help prevent issues like this from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Misc-static-assert | Objective Compiler |
| CodeSonar | 7.1.p0 | (customization) | Static Analysis tool for SAST solutions |
| Éclair | 1.2 | CC2.DCL03 | Static Checker |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 | Static Code analysis tool for next generation analysis using MISRA. |

The tools for this standard will be straightforward. Static code checkers will check potential unhandled use of a static assertion to test the value of a constant expression.This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| 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.**

| 1. Validate Input Data- All input data to a program, user or not, should be sanitized and searched  through for any unwanted input into a program that might cause unwanted results due to code.  2. Keep It Simple- Code should be straightforward and structed in such a way as to not confuse the programmer or others. Use correct naming conventions and indentation, and proper security processes. Assert tests if something is true and should not be used to test things that are not going to return true or false.  3.Sanitize Data Sent to Other Systems- Any and all data sent to other systems from this program needs to be checked  and ensure that the data is free of any possible vulnerabilities to prevent exploits to a system.  4.Use effective QA techniques- effective QA can help prevent issues like this from occurring.  5.Adopt a Secure Coding Standard- A Secure Coding standard is imperative to have when developing for a program or a platform. Coding with these policies and standards in mind can lead to more efficient and safer code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Early-catch-all | Static Code analysis tool for runtime errors. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++=ERR51 | Static Code analysis tool for next generation analysis using MISRA. |
| Parasoft | 2022.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Test Automation |
| Polyspace Bug Finder | R2022b | CERT C++: ERR51-CPP | Test Automation |

The tools for this standard will be straightforward. Static code checkers will check potential unhandled exceptions in the code. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Orient Programming | [STD-008-CPP] | Do not use pointer-to-member operators to access nonexistent members. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer-to-member object is obtained from D::g but is then upcast to be a B::\*. When called on an object whose dynamic type is D, the pointer-to-member call is well defined. However, the dynamic type of the underlying object is B, which results in undefined behavior. |
| struct B {  virtual ~B() = default;  };    struct D : B {  virtual ~D() = default;  virtual void g() { /\* ... \*/ }  };    void f() {  B \*b = new B;    // ...    void (B::\*gptr)() = static\_cast<void(B::\*)()>(&D::g);  (b->\*gptr)();  delete b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the upcast is removed, rendering the initial code ill-formed and emphasizing the underlying problem that B::g() does not exist. This compliant solution assumes that the programmer's intention was to use the correct dynamic type for the underlying object. |
| struct B {  virtual ~B() = default;  };    struct D : B {  virtual ~D() = default;  virtual void g() { /\* ... \*/ }  };    void f() {  B \*b = new D; // Corrected the dynamic object type.    // ...  void (D::\*gptr)() = &D::g; // Moved static\_cast to the next line.  (static\_cast<D \*>(b)->\*gptr)();  delete b;  } |

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

| 1. Validate Input Data- All input data to a program, user or not, should be sanitized and searched  through for any unwanted input into a program that might cause unwanted results due to code. Undefined behavior can be potentially exploited.  3.Sanitize Data Sent to Other Systems- Any and all data sent to other systems from this program needs to be checked  and ensure that the data is free of any possible vulnerabilities to prevent exploits to a system. Undefined behavior can have unintended results.  4.Use effective QA techniques- effective QA can help prevent issues like this from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2022b | CERT C++: OOP55-CPP | Test Automation |
| Parasoft | 2022.1 | CERT\_CPP-OOP55-a  Runtime Detection | Test Automation |
| Klocwork | 2022.3 | CERT.OOP.PTR\_MEMBER\_NO\_MEMBER | Static Code analysis tool |
| CodeSonar | 7.1.p0 | LANG.MEM.UVAR | Static Analysis tool for SAST solutions |

The tools for this standard will be straightforward. Static code checkers will check for the use of pointer-to-member operators to access nonexistent members in the code. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | [STD-009-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| A std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #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** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #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.**

| 1.Adhere to the Principle of Least Privilege- Principle of least privilege is when a user account is only given permissions that are essential to perform for its intended use. This works hand in hand with Default deny to help a program determine which user should get which permissions.  2.Use effective QA techniques- effective QA can help prevent issues like this from occurring.  3. Adopt a Secure Coding Standard- A Secure Coding standard is imperative to have when developing for a program or a platform. Coding with these policies and standards in mind can lead to more efficient and safer code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1.p0 | ALLOC.LEAK | Static Analysis tool for SAST solutions |
| Klocwork | 2022.3 | RH.LEAK | Static Code analysis tool |
| Parasoft | 2022.1 | CERT\_CPP-FIO51-a  Runtime Detection | Test Automation |
| Polyspace Bug Finder | R2022b | CERT C++: FIO51-CPP | Test Automation |

The tools for this standard will be straightforward. Static code checkers will check for unclosed files when they are no longer needed. This can be used in the and creation of DevSecOps toolchain.

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory | [STD-010-CPP] | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the local variable space is passed as the expression to the placement new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined behavior due to ::operator delete() 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** |
| --- |
| This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. This is one of the few times when explicitly invoking a destructor is warranted. |
| #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.**

| 1.Architect and Design for security Policies- it is important to know the security policies, and a memory allocation error can cause many problems if not properly handled.  2. Practice defense in depth- proper memory allocation can help improve the defense in depth.  3.Use effective QA techniques- effective QA can help prevent issues like this from occurring, and with memory allocation being very important should have a lot of attention paid to ensure this does not happen. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2022b | CERT C++: MEM51-CPP | Test Automation |
| Astree | 20.10 | Dangling-pointer-use | Static Code analysis tool for runtime errors. |
| CodeSonar | 7.1.p0 | ALLOC.DF  ALLOC.FNH  ALLOC.LEAK  ALLOC.TM | Static Analysis tool for SAST solutions |
| Parasoft | 2022.1 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d  Runtime Detection | Test Automation |

The tools for this standard will be straightforward. Static code checkers will check for Properly deallocated dynamically allocated resources. If resources are not properly allocated, warnings should be trown. This can be used in the and creation of DevSecOps toolchain.

### 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 very important to the development process of a program. Automation should be included during the verifying and testing and even after the release of the program because the of how helpful it is to constantly improve upon a product constantly, even after the initial release.

### 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 | High | 2 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is when data is written into storage on a disk and having the data encrypted. If an attacker manages to get the drive without the keys, they will be unable to read the data. This policy apples to this process because it allows users to safeguard their data when not in use. |
| Encryption at flight | Encryption if flight is when data is being transmitted from one source to another in some way, often over a network. It is important because encrypting data and ensuring that the users have the proper keys to access the data is important. |
| Encryption in use | Encryption is use is the method of protecting data so that it is never unsecured, regardless of its lifecycle stage (at rest, in transit), source, or location. This is important to every step of the stage in DevSecOps because it will protect the user at every step of the way if followed correctly. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Triple AAA Authentication describes the network or application way of identifying a user and making sure that the user is who they claim to be. This can involve user logins with usernames and passwords. Authentication ensures proper authorization to access a system. Generally, the system compares the users credentials with other credentials stored in the database. |
| Authorization | Triple AAA authorization is the method that is used to enforce policies like determining the activities, services and resources that a user is allowed to use. Users are assigned authorization levels that define their levels of access to a network that dictate what actions a user can do. This can be based on different restrictions, such as geolocation, frequency of logins, etc. This encompasses user levels of access, and addition of new users, as well as changes to the database. |
| Accounting | Triple AAA Accounting is the measure of resources that a user consumes during access to a network/application. This information is used for authorization control, billing, trend analysis. This ensures that an audit will enable administrators to login and view actions performed by whoever at what time. |

**\***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. Keep it Simple- Logs should be straightforward and uncomplicated.

> 5. Default Deny- Access is restricted by default until given access.

> 6. Adhere to the Principle of Least Privilege- Only needed permissions are given out to users.

> 9. Use Effective Quality Assurance Techniques- Proper QA will ensure that the system logs are clean.

> 10. Adopt a secure coding standard- ensure that the operating system logs insure that they are adhering to the Coding Standard set forth in policy.

* Firewall logs

> 4. Keep it Simple- Logs should be straightforward and uncomplicated.

> 5. Default Deny- Access is restricted by default until given access.

> 6. Adhere to the Principle of Least Privilege- Only needed permissions are given out to users.

> 8. Defense in Depth- firewall logs are part of the defense in depth with other parts.

> 10. Adopt a secure coding standard- ensure that the operating system logs insure that they are adhering to the Coding Standard set forth in policy.

* Anti-malware logs

> 4. Keep it Simple- Logs should be straightforward and uncomplicated.

> 5. Default Deny- Access is restricted by default until given access.

> 6. Adhere to the Principle of Least Privilege- Only needed permissions are given out to users.

> 8. Defense in Depth- firewall logs are part of the defense in depth with other parts.

> 10. Adopt a secure coding standard- ensure that the operating system logs insure that they are adhering to the Coding Standard set forth in policy.

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 | 10/15/2022 | Project One | Eric Ginder | Eric Ginder |
| [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 |