**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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 21](#_Toc52464067)

[Coding Standard 10 23](#_Toc52464068)

[Defense-in-Depth Illustration 25](#_Toc52464069)

[Project One 25](#_Toc52464070)

[1. Revise the C/C++ Standards 25](#_Toc52464071)

[2. Risk Assessment 25](#_Toc52464072)

[3. Automated Detection 25](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 26](#_Toc52464076)

[7. Map the Principles 27](#_Toc52464077)

[Audit Controls and Management 30](#_Toc52464078)

[Enforcement 30](#_Toc52464079)

[Exceptions Process 30](#_Toc52464080)

[Distribution 31](#_Toc52464081)

[Policy Change Control 31](#_Toc52464082)

[Policy Version History 31](#_Toc52464083)

[Appendix A Lookups 31](#_Toc52464084)

[Approved C/C++ Language Acronyms 31](#_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 | Input data validation is one of the first steps to ensuring that your code is secure. It is important to anticipate that an attacker may decide to give input into a system that will cause unexpected behaviors. Validating input data will help to prevent this from happening and could serve to check when a would-be attacker is attempting to hack your system. |
| 1. Heed Compiler Warnings | Compiler warnings are very useful tools to help you determine how your code may act given certain circumstances. A compiler will likely tell you if a library function has been deprecated to increase security and quality, which would be a very important warning to heed. It can also predict when some code will have unexpected behaviors based on the data types or built-in functions used and how they are used in the code. |
| 1. Architect and Design for Security Policies | Having a strong security policy is a good way to ensure that writing secure and efficient code is a focus when beginning the development lifecycle. It will also help to create a standard for an organization to adhere to that will increase the likelihood that everyone who writes code will write it with the same level security every time. While it is not guaranteed to produce perfectly secure code, it is a great place to start and will lead to more secure code as the policies are revised and perfected. |
| 1. Keep It Simple | It is important to keep code as simple as possible to complete the desired tasks. The more complicated that code becomes, the more difficult it will be to check for security flaws. Code that is simple is easier to understand, and therefore, it will be easier to spot security flaws while coding and during testing. |
| 1. Default Deny | Default Deny is a way to ensure that no unwanted intruders can make it into a system. It is a strategy that denies access to a system that any user or data traffic that is not explicitly allowed as necessary. It keeps users who do not have a need to access the system or its data from being able to get in and wreak havoc on the system. |
| 1. Adhere to the Principle of Least Privilege | One of the simplest and most widely used secure coding strategies is the Principle of Least Privilege. This strategy only allows users to have access to data and functionality that is immediately necessary to perform the tasks that they need to perform. Everything else, no matter the level of security or importance, is locked away. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to another system is the process in which data is permanently deleted from the original system when it is sent to another system. This can help to ensure that important or secret data is not accessible to attackers if they are able to infiltrate a system. It is also a good way to ensure that no duplicate data is left to be forgotten in a system that is no longer monitored, where an attacker will have an easy time of accessing the data. |
| 1. Practice Defense in Depth | While each secure coding strategy is powerful and effective on its own, it is vital that any relevant strategy be implemented when creating code. The more security measures taken to ensure that code does not contain security flaws, the less likely an attacker will be able to infiltrate the system. Creating multiple layers of security may seem like a daunting task; however, it is worth it to ensure that code written for a system is properly secure. |
| 1. Use Effective Quality Assurance Techniques | Just as security strategies are important to ensure secure code, effective quality assurance techniques are important to finding any security flaws that may have slipped through the cracks as the code was being developed. It is important that good and effective quality measures be taken to ensure that any rogue security flaws are weeded out and tested for. This will help to raise confidence in the system and its ability to keep unwanted intruders out. |
| 1. Adopt a Secure Coding Standard | Every organization must adopt a secure coding standard. This will allow for security flaws to be found more easily as the standard is continually carried out. It will also help for secure coding to be more of a focus and allows for repeatable code security and quality, since the standard would be applied across the entire organization. |

### 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** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **Data Type** | [STR-050-CPP] | When trying to place input data into a string that does not have enough space for the data can result in buffer overflows. |

| **Noncompliant Code** |
| --- |
| This code block shows code that does not check for input data length before placing data into a bounded string array. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best way to make sure that data is not truncated is to use the C++ std::string instead of an array of char. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Validate Input Data – validating input will help to ensure that any data input into a string will fit into the bounds of the variable, thus helping to ensure that a buffer overflow does not occur. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stream-input-char-array | Partially checked + soundly supported |
| CodeSonar | 7.4p0 | MISC.MEM.NTERM | No space for null terminator |
| Helix QAC | 2023.3 | C++5216 |  |
| Klocwork | 2023.3 | NNTS.MIGHT |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | [INT-050-CPP] | The range of values represented by scoped and unscoped enumerations may include values not specified by the enumeration. |

| **Noncompliant Code** |
| --- |
| This code block checks to see if the values are within range, but only after it has been cast to the enumeration type. |
| 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 code block checks that the value is within the range of the enumeration type before casting it to the enumeration type. |
| 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.**

| **Principles(s):** Use Effective Quality Assurance Techniques – Using bounds checking where effective is an important quality assurance technique that will help to ensure that no out of bounds values will be used. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Cast-integer-to-enum | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CERTC++-INT50 |  |
| CodeSonar | 7.4p0 | LANG.CAST.COERCE | Coercion Alters Value |
| Helix QAC | 2023.3 | C++3013 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Use valid references, pointers, and iterators to reference elements of a basic\_string** |
| --- | --- | --- |
| **String Correctness** | [STR-052-CPP] | Since a basic\_string is an array of characters, using invalidated references, pointers, and iterators could result in unexpected behavior. |

| **Noncompliant Code** |
| --- |
| This code block shows an example of a copy into a std::string where the variable “loc” becomes invalidated after the first call to insert(). |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

| **Compliant Code** |
| --- |
| This example shows how the variable “loc” is updated from each call to insert() that does not cause it to become invalidated. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

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

| **Principles(s):** Architect and Design for Security Policies – Ensuring the use of valid references can help to mitigate undefined behaviors that will make it more difficult for attackers to exploit vulnerabilities in the program. Focusing on this from the beginning will help to mitigate issues that could arise from this example in other areas of the code as well. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | ALLOC.UAF | User After Free |
| Helix QAC | 2023.3 | DF4746, DF4747, DF4748, DF4749 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2023a | CERT C++: STR52-CPP | Checks for use of invalid string iterator (rule partially covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [IDS-000-J] | SQL Injections can allow attackers to retrieve sensitive data from a system. |

| **Noncompliant Code** |
| --- |
| This coding example allows a SQL Injection attack by using unsanitized data, allowing an attacker to use the common attack “valid user or ‘1=1’”. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;    class Login {  public Connection getConnection() throws SQLException {  DriverManager.registerDriver(new  com.microsoft.sqlserver.jdbc.SQLServerDriver());  String dbConnection =  PropertyManager.getProperty("db.connection");  // Can hold some value like  // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  return DriverManager.getConnection(dbConnection);  }    String hashPassword(char[] password) {  // Create hash of password  }    public void doPrivilegedAction(String username, char[] password)  throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  try {  String pwd = hashPassword(password);    String sqlString = "SELECT \* FROM db\_user WHERE username = '"  + username +  "' AND password = '" + pwd + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString);    if (!rs.next()) {  throw new SecurityException(  "User name or password incorrect"  );  }    // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  }  }  } |

| **Compliant Code** |
| --- |
| This code example uses a parametric query with “?” as a place holder for the argument, as well as validating the length of the username to prevent an attacker from submitting arbitrarily long values. |
| public void doPrivilegedAction(  String username, char[] password  ) throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  try {  String pwd = hashPassword(password);    // Validate username length  if (username.length() > 8) {  // Handle error  }    String sqlString =  "select \* from db\_user where username=? and password=?";  PreparedStatement stmt = connection.prepareStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, pwd);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("User name or password incorrect");  }    // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – SQL Injection is one of the most common forms of software vulnerability. Preparing for and preventing SQL Injections is one of the first steps to adopting a secure coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 7.4p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Coverity | 7.5 | SQLI | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Protection** | [MEM-051-CPP] | Passing a pointer value to an inappropriate deallocation function could cause the program to exhibit unexpected behavior. |

| **Noncompliant Code** |
| --- |
| This coding example shows an attempt to free resources in a catch block. It does not, however, initialize the variables, allowing the “i1” variable to pass ::operator delete() to “i2” that was not returned by a call to ::operator new(). |
| #include <new>    void f() {  int \*i1, \*i2;  try {  i1 = new int;  i2 = new int;  } catch (std::bad\_alloc &) {  delete i1;  delete i2;  }  } |

| **Compliant Code** |
| --- |
| In this coding example, both pointer values are initialized to “nullptr”, which is a valid value to pass to ::operator delete(). |
| #include <new>    void f() {  int \*i1 = nullptr, \*i2 = nullptr;  try {  i1 = new int;  i2 = new int;  } catch (std::bad\_alloc &) {  delete i1;  delete i2;  }  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Properly deallocating dynamically allocated resources is a common security practice and should be well implemented in a coding standard as a quality assurance technique to ensure it is properly handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Invalid\_dynamic\_memory\_allocation |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM51 |  |
| Clang | 3.9 | Clang-analyzer-cplusplus.NewDeleteLeaks | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 7.4p0 | ALLOC.FNH | Free non-heap variable |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [DCL-003-C] | The runtime assert() macro is only useful for identifying incorrect assumptions and not runtime error checking. Static\_assert() will produce an error message, set by the programmer, at runtime if the constant\_expression is false. |

| **Noncompliant Code** |
| --- |
| Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed. |
| #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** |
| --- |
| Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Because the assertion is performed at compile time, no runtime cost in space or time is incurred. An assertion can be used at file or block scope, and failure results in a meaningful and informative diagnostic error message. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Assertions are a common quality testing practice. Using static assertions will help with quality testing of constant expressions, since constant expressions do not change. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.4p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [ERR-051- CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. |

| **Noncompliant Code** |
| --- |
| In this coding 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 coding example, 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.**

| **Principles(s):** Architect and design for security policies – Unhandled exceptions are a good opportunity for attackers to infiltrate a system and steal data or wreak havoc on the system. It is important to focus on security and ensure all potential exceptions are handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Mian-function-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 7.4p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2023.3 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not read uninitialized memory** |
| --- | --- | --- |
| Expressions | [EXP-053- CPP] | Local, automatic variables assume unexpected values if they are read before they are initialized. |

| **Noncompliant Code** |
| --- |
| In this coding example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this example, the object is initialized prior to printing its value. |
| #include <iostream>    void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** Heed compiler warnings – Most compilers will issue a warning about using uninitialized variables. It is important to heed these warnings as they can lead to undefined behavior that can be exploited by attackers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Uninitialized-read | Partially checked |
| Clang | 3.9 | -Wuninitialized | Does not catch all instances of this rule, such as uninitialized values read |
| CodeSonar | 7.4p0 | LANG.STRUCT.RPL | Return pointer to local |
| Helix QAC | 2023.3 | DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF2978 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Write Constructor member initializers in canonical order** |
| --- | --- | --- |
| Object Oriented Programming | [OOP-053- CPP] | The member initializer list for a class constructor allows members to be initialized to specified values and for base class constructors to be called with specific arguments. However, the order in which initialization occurs is fixed and does not depend on the order written in the member initializer list. |

| **Noncompliant Code** |
| --- |
| In this coding example, the member initializer list for C::C() attempts to initialize someVal first and then to initialize dependsOnSomeVal to a value dependent on someVal. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of someVal results in an unspecified value being stored into dependsOnSomeVal. |
| class C {  int dependsOnSomeVal;  int someVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| This coding example changes the declaration order of the class member variables so that the dependency can be ordered properly in the constructor's member initializer list. |
| class C {  int someVal;  int dependsOnSomeVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** Keep it simple – Writing constructor initializers in canonical order helps to keep constructor’s initializers organized and will help those who are reviewing the code to better understand when each is used, thus simplifying the code and making it more readable and easier to follow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Initializer-list-order | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP53 |  |
| Clang | 3.9 | -Wreorder |  |
| CodeSonar | 7.4p0 | LANG.STRUCT.INIT.OOMI | Out of Order Member Initializers |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Wrap functions that can spuriously wake up in a loop** |
| --- | --- | --- |
| Concurrency | [CON-054- CPP] | The wait(), wait\_for(), and wait\_until() member functions of the std::condition\_variable class temporarily cede possession of a mutex so that other threads that may be requesting the mutex can proceed. These functions must always be called from code that is protected by locking a mutex. |

| **Noncompliant Code** |
| --- |
| This coding example nests the call to wait() inside an if block and consequently fails to check the condition predicate after the notification is received. If the notification was spurious or malicious, the thread would wake up prematurely. |
| #include <condition\_variable>  #include <mutex>    struct Node {  void \*node;  struct Node \*next;  };    static Node list;  static std::mutex m;  static std::condition\_variable condition;    void consume\_list\_element(std::condition\_variable &condition) {  std::unique\_lock<std::mutex> lk(m);    if (list.next == nullptr) {  condition.wait(lk);  }    // Proceed when condition holds.  } |

| **Compliant Code** |
| --- |
| This coding example calls the wait() member function from within a while loop to check the condition both before and after the call to wait(). |
| #include <condition\_variable>  #include <mutex>    struct Node {  void \*node;  struct Node \*next;  };    static Node list;  static std::mutex m;  static std::condition\_variable condition;    void consume\_list\_element() {  std::unique\_lock<std::mutex> lk(m);    while (list.next == nullptr) {  condition.wait(lk);  }    // Proceed when condition holds.  } |

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

| **Principles(s):** Keep it simple – Wrapping functions that could spuriously wake up in a loop in a while loop, as opposed to a for loop, will allow for better condition handling that can minimize needed code in other sections to activate or deactivate the wait function. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.STRUCT.ICOL | Inappropriate Call Outside Loop |
| Helix QAC | 2023.3 | C++5019 |  |
| Klocwork | 2023.3 | CERT.CONC.WAKE\_IN\_LOOP |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CON54-a | Wrap functions that can spuriously wake up in a loop |

### 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 with 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 should take place in the “Verify and Test” phase of the DevOps process. This will allow it to speed up the testing process, as well as fit into the natural process of compliancy and security testing. Also, this is the phase at which automation should be the least intrusive when adding it into the process, to minimize the need for a transition period to ensure that there is no confusion and fewer mistakes are made from adding in new steps to the process.

### 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 |
| --- | --- | --- | --- | --- | --- |
| CON-054-CPP | Low | Unlikely | Medium | P2 | L2 |
| DCL-003-C | Low | Unlikely | High | P1 | L3 |
| ERR-051-CPP | Low | Probable | Medium | P4 | L3 |
| EXP-053-CPP | High | Probable | Medium | P12 | L1 |
| IDS-000-J | High | Likely | Medium | P18 | L1 |
| INT-050-CPP | Medium | Unlikely | Medium | P4 | L3 |
| MEM-051-CPP | High | Likely | Medium | P18 | L1 |
| OOP-053-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STR-050-CPP | High | Likely | Medium | P18 | L1 |
| STR-052-CPP | High | Probable | High | P6 | L2 |

### 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 they are, how they should be applied in practice, and why they should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | This type of encryption is used when data is being stored, whether on a disk drive or in a database. It should be used when storing data on servers or in databases by using symmetric encryption. |
| Encryption in flight | This type of encryption is used for data that is in transit. It should be used when sending or receiving data, whether over wired or wireless networks, using Secure Socket Layer level encryption. |
| Encryption in use | This type of encryption is used when data is being actively used, it also helps with analyzing data requests and denies access to unauthorized users. This type of encryption would be used when users are using data for analysis or to perform their work tasks by using homomorphic encryption. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is used to verify that a user has been granted access to the system in use, using user IDs and passwords. This keeps unwanted and unauthorized users from accessing sensitive data within the system. |
| Authorization | Authorization is the level of access that each authorized user has in the system. This should follow the principle of least privilege to ensure that no one can access data that has no direct need of it. This will help to ensure that authorized users cannot make changes to data or databases and that no user without proper authorization can add or remove other users. |
| Accounting | Accounting is the act of monitoring the usage of data and the level of authorization that other users have within the system. It is important to keep an accurate account of who is using the system, what they are using it for, and when they access the system. This can help to mitigate intrusions, as well as quickly find where an unauthorized access occurred. |

**\***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
* Coding Standard 1:

1: Coding standard 1 uses user input to populate data to a variable. It is always important to validate user input to avoid undefined behavior and potential threats from attackers.

3: Ensuring that variables have sufficient space for data received from a user is one of the first security measures for coding that is taught to up-and-coming programmers.

9: Checking the length of data given by a user is a great step to ensuring that the quality of the program being developed is at an acceptable level.

* Coding Standard 2:

2: Casting to an out-of-range value will result in the compiler issuing an ‘out-of-bounds’ warning. These warnings can be catastrophic and should be fixed when encountered.

4: It can help to keep the coding simple to prevent from accidentally attempting to access a range out of bounds from an enumeration type.

9: Checking for out-of-bounds errors or potential errors is an important quality assurance technique. It will keep flaws from causing a program to crash or allowing an attacker to infiltrate a system.

* Coding Standard 3:

2: Compilers will likely give warnings if the references, pointers, or iterators to a basic\_string are not valid. It is important to fix these as there may be undefined behavior.

3: Ensuring that references to a basic\_string are valid is important to the security of the program. If these are not valid, it could allow attackers to obtain sensitive data or gain unauthorized access to the system.

8: This coding standard is an important one to keep in mind when practicing defense in depth. It will help to improve the security of the system and ensure that data is secured, and the system cannot be accessed by unauthorized users.

* Coding Standard 4:

1: Validating user input is the first step in preventing and detecting potential SQL Injection attacks. It can stop the input values from having an unexpected effect on the system, preventing an attacker from accessing the system.

3: SQL Injections are a common form of attack on systems that contain sensitive data in databases or servers. Having a strong security policy will help to mitigate the likelihood of SQL Injections on the system.

10: When adopting a secure coding standard, it is important to remember the most common types of attacks. This can ensure that programmers do not get comfortable in forgetting about them, and can build onto security standards for other, lesser-known types of attacks.

* Coding Standard 5:

3: It is important to make sure that all dynamically allocated resources are properly deallocated. This can help to ensure that these resources do not cause potential leaks in memory or vulnerabilities that can be exploited by attackers.

8: Ensuring that dynamic resources have been properly deallocated can help to prevent memory leaks, using this standard with others will help to develop a robust defense in depth structure to help improve the security of the system.

9: Checking for the proper deallocation of resources is an important quality assurance technique that can help to improve the overall quality of the system being developed. It will help to ensure that memory leaks are less likely, improving overall performance and security.

* Coding Standard 6:

8: Using static assertion tests on constant expressions is a useful tool to ensure that new code that may interact with the constant expression does not develop unexpected results. This is an important factor of defense in depth.

9: Static assertions can be used to check the results of constant expressions against other parts of the system that may interact with it. Doing this will help to develop effective quality assurance techniques.

* Coding Standard 7:

1: When taking user input for a value, it is important to check that the input is relevant and compatible with the variable that is taking that input. Not handling such an exception could result in a system crashing or an exploitable vulnerability.

3: Exceptions are common in programming a system. It is important to design security policies to handle exceptions to ensure that all exceptions within a reasonable amount are handled.

* Coding Standard 8:

2: Attempting to read uninitialized memory will likely result in a compiler warning. Fixing such issues will help to prevent vulnerabilities and coding flaws that could potentially cause problems.

3: Designing security policies to support initializing memory before it is used is an important step in creating a complete security policy to ensure that undefined behavior is minimized as much as possible.

4: Initializing memory when it is declared can help to remove the need to initialize the variable when it is to be used. This will help to simplify code and make it easier to read.

* Coding Standard 9:

4: Writing constructor member initializers in order of which they appear will help to improve readability and will help to simplify the code and make it easier for readers to understand where to find each constructor member in the code.

* Coding Standard 10:

4: Wrapping code that can spuriously wake up in a loop can help to alleviate the need for exception handling, which will help to simplify the code further.

8: This coding standard can help to improve an already strong defense in depth structure by ensuring that code that could wake up in a loop is properly handled and cannot be exploited.

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 the 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 | 09/17/2023 | Security Principles | Ryne Williams | [Insert text.] |
| 1.2 | 10/06/2023 | Coding Standards | Ryne Williams | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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