**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 | When receiving input data, it is important to validate all input data from untrusted sources. It is important to do this, because untrusted sources can provide input that is incorrect, in an incorrect format, or doesn’t meet the program’s input constraints that may put a program’s security at risk. With proper input data validation, the program checks the input received which can help eliminate software vulnerabilities. For example, a programmer can implement methods to check for proper data type entry, such as only accepting integers, and character length to help prevent an SQL Injection attack or overflow buffer attack. |
| 1. Heed Compiler Warnings | Heeding compiler warnings includes the use of static and dynamic tools using the highest warning level available to detect and eliminate security flaws within a program by modifying the code within the program. The compiler can check for flaws within methods of a program and check for flaws that might be used by a malicious hacker. If a flaw is detected, the compiler can proactively warn and even correct flawed methods to a more safe application. |
| 1. Architect and Design for Security Policies | When building and designing the architecture for a software program, it is important to create the design around security policies and implementing them directly within the architecture’s design. By designing security policies into the architecture of a program, software vulnerabilities can be reduced or eliminated. Some ways that this can be done include implementing privileges and permissions to users based upon their role and only giving them access to just enough information that they need to access and not anymore, create different levels of security within the program, and adding safeguards and encryptions to protect sensitive data. |
| 1. Keep It Simple | When designing software, keep the design as simple and small as possible, because more complex designs are more likely to have errors within implementation, configuration, and use of the software. In addition, more complex software programs require more complex security measures than simpler programs. Therefore, it is best to keep the design as simple and small as possible. |
| 1. Default Deny | Default deny is a method where access is denied by default to all users and only providing access to users under circumstances where permission is granted. In other words, default deny only provides access to certain users with special permissions in order to protect the program’s sensitive data from being accessed by unauthorized users. For example, a customer of a business should have less access than an employee, therefore both users would by default be denied access to the program’s information. Once the employee enters their credentials and the system verifies the specific conditions against their access, the employee could then access the information. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the principle of least privilege includes only giving users the access that they require to complete necessary work or tasks without giving them access to too much information that they don’t require. In addition to only allowing them access to information they need, they should only be allowed to access the information or permissions for a limited amount of time required for them to complete those tasks. This helps to reduce the number of opportunities for an attacker to access elevated permissions. For example, a company may allow certain employees, such as store managers, access to financial reports by allowing them permissions to pull reports within a certain timeframe. |
| 1. Sanitize Data Sent to Other Systems | When data is sent to other complex subsystems, the data should be sanitized first before it is passed along to a subsystem where a hacker can invoke functionality that isn’t used or used less frequently. Therefore, the calling method should validate and sanitize data received before passing it along to a subsystem which could then become vulnerable. To help sanitize data, input validations, output encoding, and running queries of databases with safe, prepared queries instead of user inputs should be used. |
| 1. Practice Defense in Depth | A software program should practice defense in depth, or multiple defense tactics at different layers within the program to prevent security vulnerabilities. By using multiple defense tactics, this can help mitigate threats because if one layer fails then another layer can help prevent that security flaw from being used by a hacker. For example, to implement and practice defense in depth, an organization can implement security at different levels, such as implement firewalls for their network, strong user authorization controls, encrypt sensitive data, and implement security principles, such as the principle of least privilege or designing and architecting on security policies. |
| 1. Use Effective Quality Assurance Techniques | When a program is tested to identify and eliminate vulnerabilities, the use of effective quality assurance techniques is important to ensure that a thorough evaluation of the program is completed. Effective quality assurance techniques aim to identify vulnerabilities through testing and eliminate those vulnerabilities before the final version of the program is released where a hacker could find and exploit a vulnerability. Examples of quality assurance techniques include penetration testing, source code audits, and code reviews and analyses. |
| 1. Adopt a Secure Coding Standard | A company should adopt a secure coding standard that their business adheres to integrate security into new and existing systems. Adopting a secure coding standard includes adhering to a well-known standard and outlining guidelines for secure coding, which can include avoiding known vulnerabilities, implementing security best practices, and using thorough testing processes. The company then should promote, educate, and implement these secure coding standards within their development teams to produce code that is secure and functional. |

### 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** | **Obey the One-Definition Rule** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | The [obey the one-definition rule coding standard](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule) states that every program shall contain exactly one definition for every functional or variable used within the program. |

| **Noncompliant Code** |
| --- |
| In this example, noncompliant code is shown with two different translation units that are defining a class of the same name with different definitions. |
| // x.cpp  struct J {    int z;  };    // y.cpp  class J {  public:    int z;  }; |

| **Compliant Code** |
| --- |
| In this corrected and compliant code, if the developer wants the same class definitions to be visible in both of the translation units, then they will need to use a header file to introduce the object into the translation units. |
| // Z.h  struct G {    int a;  };    // x.cpp  #include "Z.h"    // y.cpp  #include "Z.h" |

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

| **Principles(s):** 10. Adopt a Secure Coding Standard – This principle applies to the Data Type Coding Standard, because it is a secure coding standard to ensure that each function and variable within a program has exactly one definition to ensure that each function and variable is unique and behaves as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL60-a | The One Definition Rule shall not be violated. |
| Polyspace Bug Finder | R2025b | CERT C++: DCL60-CPP | Checks for inline constraints not respected (rule partially covered). |
| CodeSonar | 9.1p0 | LANG.STRUCT.DEF.FDH  LAND.STRUCT.DEF.ODH | Function defined in header file.  Object defined in header file. |
| LDRA Tool Suite | 9.7.1 | 286 S, 287 S | Fully implemented. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do Not Write Syntactically Ambiguous Declarations** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | The second standard: [Do Not Write Syntactically Ambiguous Declarations](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL53-CPP.+Do+not+write+syntactically+ambiguous+declarations), states that it’s possible to develop syntax that can be ambiguously interpreted as an expression statement or a declaration. This results in syntax that is called a vexing parse, which is where the compiler must use disambiguation rules to decide the correct result. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an attempt is made to declare a local variable, w, of type Widget while executing the default constructor, but the declaration is syntactically ambiguous where the code could either be a declaration of a function pointer that accepts no arguments and returning a Widget or a declaration of a local variable of type Widget. |
| #include <iostream>    struct Widget {    Widget() { std::cout << "Constructed" << std::endl; }  };    void f() {    Widget w();  } |

| **Compliant Code** |
| --- |
| In this compliant code example that has been corrected, the parentheses are removed after the variable declaration to ensure that the syntax is of a variable declaration rather than a function declaration. |
| #include <iostream>    struct Widget {    Widget() { std::cout << "Constructed" << std::endl; }  };    void f() {    Widget w1; // Elide the parentheses    Widget w2{}; // Use direct initialization  } |

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

| **Principles(s):** 2. Heed Compiler Warnings – By heeding compiler warnings, a complier can detect syntactically ambiguous declarations that have been made and warn the developer. The developer should then heed the compiler’s warnings and refactor the code to prevent this from occurring.  10. Adopt A Secure Coding Standard – The Adopt a Secure Coding Standard applies to the Data Value coding standard, because it is important to follow best secure coding standards and coding best practices to ensure all definitions are accurate and don’t result in unintended or undefined behaviors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |
| LDRA Tool Suite | 9.7.1 | 296 S | Partially implemented. |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b  CERT\_CPP-DCL53-c | Parameter names in function declarations should not be enclosed in parentheses.  Local variables names in variable declarations should not be enclosed in parentheses.  Avoid function declarations that are syntactically ambiguous. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do Not Attempt to Create a std::string From a Null Pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The [Do Not Attempt to Create a std::string From a Null Pointer](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer) standard is defined as the std::basic\_string type utilizes traits design patterns to handle implementation details of various string types, which results in a series of string-like classes with a common, underlying implementation. According to the C++ standard, passing a null pointer to the std::char\_triats::length() function is undefined behavior because it would dereference a null pointer. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::genenv(), but because std::getenv() returns a null pointer on failure, the code leads to undefined behavior where the environmental variable does not exist, or another error can occur. |
| #include <cstdlib>  #include <string>    void f() {    std::string tmp(std::getenv("TMP"));    if (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this complaint code example, 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.**

| **Principles(s):** 2. Heed Compiler Warnings – The principle of Heed Compiler Warnings applies to the String Correctness Coding Standard, because it can warn when there is a potential instance of creating a std::string from a null pointer. By heeding the compiler’s warning, a developer can prevent this issue from leading to undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2025b | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |
| Security Reviewer – Static Reviewer | 6.02 | shiftTooManyBits | Fully implemented. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-JAV] | The [Prevent SQL Injection](https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection) standard states that untrusted data can maliciously alter a query, which can result in information leaks or data modification. The primary way to prevent an SQL injection attack are sanitizing and validating data received as an input. |

| **Noncompliant Code** |
| --- |
| The JDBC library provides an API for building SQL commands that sanitize untrusted data and the java.sql.PreparedStatement class properly escaped input strings, which can prevent SQL injection attacks. In this noncompliance code example, the code modified the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement, but the prepared statement still permits an SQL injection attack by using unsanitized input for the username into the prepared statement. |
| 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;        PreparedStatement stmt = connection.prepareStatement(sqlString);          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        }      }    }  } |

| **Compliant Code** |
| --- |
| In this corrected and compliant code example, the solution uses a parametric query with a ? character as a placeholder for the argument. This validates the length of the username argument, which prevents an attacker from entering a username that is too long. |
| 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):** 1. Validate Input Data – The Validate Input Data principle applies to the SQL Injection Coding Practice, because it is best practice to validate input data received by users before inputting it into a program, which could result in a SQL injection attack if the input received isn’t validated. It’s best practice to never trust a user’s input, but instead to check the input received for correctness.  7. Sanitize Data Sent to Other Systems – The Sanitize Data Sent to Other System also applies to the SQL Injection Coding Practice, because a hacker can pass malicious information to other systems that can become vulnerable to an attack. It is best practice to validate input received and sanitize it before passing it to other systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| SonarQube | 9.9 | S2077  S3649 | Executing SQL queries is security-sensitive.  SQL queries should not be vulnerable to injection attacks. |
| Parasoft Jtest | 2024.2 | CERT.IDS00.TDSQL | Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Provide Placement New with Properly Aligned Pointers to Sufficient Storage Capacity** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | In the fifth standard, [Provide Placement New with Properly Aligned Pointers to Sufficient Storage Capacity](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM54-CPP.+Provide+placement+new+with+properly+aligned+pointers+to+sufficient+storage+capacity), when invoked by a new expression for a given type, the default global non-placement forms of operator new attempt to allocate sufficient storage for an object of the type, and if successful, return a pointer with alignment suitable for any object with a fundamental alignment requirement. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the long is constructed into a buffer of sufficient size but does not ensure that the alignment requirements are met for the pointer passed into placement new. |
| #include <new>    void f() {    char c; // Used elsewhere in the function    unsigned char buffer[sizeof(long)];    long \*lp = ::new (buffer) long;      // ...  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the alignas declaration specified is used to ensure the buffer is appropriately aligned for a long. |
| #include <new>    void f() {    char c; // Used elsewhere in the function    alignas(long) unsigned char buffer[sizeof(long)];    long \*lp = ::new (buffer) long;      // ...  } |

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

| **Principles(s):** 10. Adopt a Secure Coding Standard – The Adopt a Secure Coding Standard principle applies to the Memory Protection Coding Standard, because it is best practice to allocate enough memory for any objects within a program. Therefore, it is important to allocate enough memory for each object and to check there is enough memory allocated before implementing new expressions, functions, and pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | LANG.MEM.BO | Buffer Overrun |
| LDRA Tool Suite | 9.7.1 | 597 S | Enhnced Enforcement |
| Polyspace Bug Finder | R2025b | CERT C++: MEM54-CPP | Checks for placement new used with insufficient storage or misaligned pointers (rule fully covered). |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Expressions Used in Assertions Must Not Produce Side Effects** |
| --- | --- | --- |
| **Assertions** | [STD-006-JAV] | In the sixth standard of [Expressions Used in Assertions Must Not Produce Side Effects](https://wiki.sei.cmu.edu/confluence/display/java/EXP06-J.+Expressions+used+in+assertions+must+not+produce+side+effects), this standard states that the assert statement is a tool for incorporating diagnostic tests in code and the behavior of the statement depends on its status of a runtime property. When the assert statement is enabled, the statement evaluated its expression argument and throws an AssertionError if false. If the assert statement is disabled, the assert statement is a no-op and any side effects results from the evaluation of the expression in the assertion are lost. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the code attempts to delete all null names from the list in the assertion statement, but the Boolean expression does not evaluate when assertions are disabled. |
| private ArrayList<String> names;    void process(int index) {    assert names.remove(null); // Side effect    // ...  } |

| **Compliant Code** |
| --- |
| In this corrected and compliant code example, the possibility of side effects in assertions is to be avoided by decoupling the Boolean expression from the assertion. |
| private ArrayList<String> names;    void process(int index) {    boolean nullsRemoved = names.remove(null);    assert nullsRemoved; // No side effect    // ...  } |

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

| **Principles(s):** 3. Architect and Design for Security Policies – The Architect and Design for Security Policies principle states that it is important to create the design of the architecture of a program around security policies and implement them directly into the architecture’s design. This principle applies to the Assertions Coding Standard, because this helps apply diagnostic testing into the code and checks its behaviors.  9. Use Effective Quality Assurance Techniques – The Use Effective Quality Assurance Techniques principle also applies to the Assertions Coding Standard, because it states that effective quality assurance technique are important to fully evaluate to identify and eliminate any vulnerabilities, including assert statements that produce side effects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | JAVA.STRUCT.SE.ASSERT | Assertion contains side effects. |
| Parasoft Jtest | 2024.2 | CERT.EXP06.EASE | Expressions used in assertions must not produce side effects. |
| SonarQube | 9.9 | S3346 | Expressions used in “assert” should not produce side effects. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle All Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | In the [Handle All Exceptions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) standard, it is stated that when an exception is thrown, control is transferred to the neared handler with a type that matches the type of the exception thrown. If there is no matching handler for a try block where an exception is thrown, the search for a matching handler continues to dynamically search for handlers in surrounding try blocks. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() or main()catch exceptions throw by throwing\_funct(),which means that std::terminate()is called instead. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the main entry point handles all exceptions, which helps to ensure 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):** 4. Keep It Simple – When designing software, it is important to keep the design as simple as possible because if the design is more complex, it can have more errors. Therefore, it is important to handle all exceptions within the definition rather than trying to handle any exceptions elsewhere in the code or using nearby handlers instead.  9. Use Effective Quality Assurance Techniques – When a program is thoroughly tested through effective quality assurance techniques, this can help identify any unhandled exceptions that may exist that need to be properly handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked. |
| CodeSonar | 9.1p0 | LANG.STRUCT.UCTCH  PARSE.MBDH | Masked by handler.  Masked by default handler. |
| RuleChecker | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked. |
| Security Reviewer – Static Reviewer | 6.02 | C35 | Fully implemented. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do Not Cast or Delete Pointers to Incomplete Classes** |
| --- | --- | --- |
| **Expressions** | [STD-008-CPP] | In the [Do Not Cast or Delete Pointers to Incomplete Classes](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP57-CPP.+Do+not+cast+or+delete+pointers+to+incomplete+classes) standard, it is defined that attempting to delete a pointer to an object of an incomplete class type can lead to undefined behavior within the program if an nontrivial destructor or deallocation function was later added. Similarly, attempting to cast through a pointer to an object of incomplete type can result in undefined behavior if the cast operating is well-formed, but the dereferencing the resulting pointer occurs due to the downcast being unable to adjust for multiple inheritance. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a class attempts to implement the pimpl idiom but instead deletes a pointer to an incomplete class type which results in undefined behavior if Body has a nontrivial destructor. |
| class Handle {    class Body \*impl;  // Declaration of a pointer to an incomplete class  public:    ~Handle() { delete impl; } // Deletion of pointer to an incomplete class    // ...  }; |

| **Compliant Code** |
| --- |
| In this corrected and compliant code example, the deletion of impl is moved to another part of the code where Body is defined. |
| class Handle {    class Body \*impl;  // Declaration of a pointer to an incomplete class  public:    ~Handle();    // ...  };    // Elsewhere  class Body { /\* ... \*/ };    Handle::~Handle() {    delete impl;  } |

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

| **Principles(s):** 10. Adopt A Secure Coding Standard – The Adopt a Secure Coding Standard principle can be applied to the Expressions Coding Standard, because it is best to adhere to well-known standards and using coding best practices to prevent undefined behavior or potential security vulnerabilities and instead has the development team complete the incomplete methods or move the pointer to a method that is fully defined. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | DELETE\_VOID | Fully implemented. |
| LDRA Tool Suite | 9.7.1 | 169 S, 554 S | Enhanced enforcement. |
| Polyspace Bug Finder | R2025b | CERT C++: EXP57-CPP | Checks for conversion or deletion of incomplete class pointer. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Guarantee That Library Functions Do Not Overflow** |
| --- | --- | --- |
| **Containers** | [STD-009-CPP] | In the [Guarantee That Library Functions Do Not Overflow](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR52-CPP.+Guarantee+that+library+functions+do+not+overflow) standard, it is stated that copying data into a container that is not large enough to hold that data can result in a buffer overflow, however data can be copied to the destination container that is restricted on the basis of the destinations container size that is guaranteed to be large enough to prevent this error. |

| **Noncompliant Code** |
| --- |
| In this noncompliance code example, a vector of integers is copied from src to dest using std::copy(), but because std::copy() does nothing to expand the dest vector, this results in a buffer overflow on the first element. |
| #include <algorithm>  #include <vector>    void f(const std::vector<int> &src) {    std::vector<int> dest;    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

| **Compliant Code** |
| --- |
| In this corrected an compliant code example, the solution enlarges the capacity of the vector prior to the operation that copies the elements to the destination. |
| #include <algorithm>  #include <vector>  void f(const std::vector<int> &src) {    // Initialize dest with src.size() default-inserted elements    std::vector<int> dest(src.size());    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

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

| **Principles(s):** 1. Validate Input Data – The Containers Coding Standard is supported by the Validate Input Data principle, because it is important to validate any input data received from a user to ensure it is the correct length and data type. By validating and sanitizing the data input received, this can help prevent overflows.  8. Practice Defense in Depth – By practicing defense in depth, an organization can encrypt their sensitive data, validate input data received, and use security measures at different levels within the system to prevent security vulnerabilities, such as overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | BADFUNC.BO.\*  LANG.MEM.BO  LANG.MEM.TBA | A collection of warning classes that report uses of library functions prone to internal buffer overflows.  Buffer overrun.  Tainted Buffer Access. |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CTR52-a | Do not pass empty container iterators to std algorithms as destinations. |
| Polyspace Bug Finder | R2025b | CERT C++: CTR52-CPP | Checks for library functions overflowing sequence container (rule partially covered). |
| Security Reviewer – Static Reviewer | 6.02 | C01  C04 | Fully implemented. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Close Files When They are No Longer Needed** |
| --- | --- | --- |
| **Input Output** | [STD-010-CPP] | The [Close Files When They are No Longer Needed](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed) standard states that a call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before a normal program termination. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the std::fstream object file is constructed and the constructor for std::fstream calls std::basic\_filebuf<T>::open(). The default terminate handler called is std::abort(), which doesn’t call destructors and the object does not properly close. |
| #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 code example, the std::fstream::close()function is called before std::terminate(), which ensures that the file resources are closed properly. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

| **Principles(s):** 3. Architect and Design for Security Policies – When designing a software program, designing the program with security into its architecture helps prevent and reduce security vulnerabilities. One way this can be done is to ensure that any files opened are also closed safely and leads to a normal program termination.  6. Adhere to the Principle of Least Privilege – The Adhere to the Principle of Least Privilege principle applies to the Input Output Coding Standard, because it should only give users access to the data for just as long as they need it. Once they have accessed the information needed, the file should close to protect the data and the program’s security.  10. Adopt a Secure Coding Standard – It is good practice to open and close all files opened within a program safely and to ensure that the program terminates normally. If a program abruptly terminates due to an error or crash, this can result is data loss. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | ALLOC.LEAK | Leak. |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-FIO51-a | Ensure resources are freed. |
| Polyspace Bug Finder | R2025b | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered). |
| Security Reviewer – Static Reviewer | 6.02 | C80 | Fully implemented. |

### 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 tools be used for the enforcement of and compliance to these standards defined in this policy to check for any security vulnerabilities and ensure that these coding policies are enforced before deployment. These tools can be used to check for vulnerabilities or non-compliant code within a program and provide reports and alerts for vulnerabilities found, policies that aren’t being followed properly, and to improve efficiency and consistency. To modify existing DevOps processes to automate enforcement of these standards, security will be incorporated into all of the pre-production stages of assess and plan, design, build, and verify and test. Security will also be incorporated into all of the production stages of transition and health check, monitor and detect, respond, and maintain and stabilize.

Within the pre-production process, automation can be incorporated into the verify and test stage. Automation can be incorporated into this stage by incorporating the use of automation test tools, which can proactively scan code for vulnerabilities, errors, and weaknesses. Automation can also test for compliance with standards by ensuring each policy is being followed correctly and won’t lead to any errors, vulnerabilities, or uncompliant code being used. With the production process, automation can be incorporated into the transition and health check and the monitor and detect stages. Automation tools can continuously run health checks of a system and run penetration testing to test for any vulnerabilities. They can also be used in the monitor and detect stage to monitor for any suspicious user activity, event alerts, logs, and track analytics for the system.

### 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 | P6 | L2 |
| STD-002-CPP | Low | Unlikely | Low | P2 | L3 |
| STD-003-CPP | High | Likely | High | P18 | L1 |
| STD-004-JAV | High | Likely | High | P18 | L1 |
| STD-005-CPP | High | Likely | High | P9 | L2 |
| STD-006-JAV | Low | Unlikely | Low | P3 | L3 |
| STD-007-CPP | Low | Probable | Medium | P6 | L2 |
| STD-008-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-009-CPP | High | Likely | High | P9 | L2 |
| STD-010-CPP | Medium | Unlikely | Medium | P2 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is a policy for protecting data by converting it to a coded format while being stored on a storage device while not in use. This policy applies encoding data by transforming it into an unreadable format until it is needed to be accessed by a user to protect the data from being accessed by an unauthorized user, prevents data from being misused or accessed, and protects it against theft. An example of encryption at rest would be the data stored in a database. |
| Encryption in flight | Encryption in flight is a policy to protect data by encrypting it to a coded format while being transmitted across a network. The data is first encrypted into an unreadable format using a key, then it is transmitted across a network, and is decrypted once it arrives at its destination. This policy applies to ensure that data remains secure while being transmitted, it remains confidential, and to maintain the integrity of the data. An example of encryption in flight would be encrypted data that is being transmitted to another user via an email. |
| Encryption in use | Encryption in use is the policy to protect data while the data is being processed or used by a user. Data is decrypted for when a user is accessing the data, but only for the duration that the data is processed or needed. Once the user is finished accessed the data, it is then encrypted to an unreadable format to protect its data. This policy applies, because it helps to protect sensitive data, even while that data is being actively used and processed. It protects this data by ensuring that the data is only decrypted for a short period of time. An example of encryption in use is browsing the internet using HTTPS. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of identifying a user to the system. This process involved comparing the user’s credentials with the stored credentials in the database to check if the username and password entered match what is known to the system. Authentication is important and applies, because it limits who can access the system and makes users identify themselves to gain access. An example of authentication includes user logins. |
| Authorization | After authentication is completed, authorization then begins to grant the users permissions based upon their role within a system. The system does this by checking the user’s identity and their role within the system, then granting them access the only the privileges they need to complete their job. This policy applies, because it authorizes what users are allowed to access. Examples of authorization include addition of new users and user level of access. |
| Accounting | Account is the process of tracking a user’s activity within the system, how long they accessed the system, and what information was accessed. This is completed by tracking each user’s movements within the system, track what information they accessed, and how long that information was accessed for and logged it in an accounting system. This policy applies, because it helps track information and holds users accountable for their actions while they were accessing the system. Examples of accounting include changes to the database, addition of new users, and files accessed by users. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 09/21/2025 | Milestone One | Jordan Crisler | Jordan Crisler |
| 3.0 | 10/09/2025 | Project One | Jordan Crisler | Jordan Crisler |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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