**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All data from sources which are not trusted must be validated. Proper input checking can help avoid several vulnerabilities. All data from external sources should be approached with suspicion. This includes command line inputs, environment variable, and any files provided. |
| 1. Heed Compiler Warnings | When code is compiled, any warnings given shall be heeded and investigated. The code will be modified such that any warnings are eliminated. In doing this the possibility of security vulnerabilities, bugs, or undefined behavior are greatly diminished. Static and dynamic analysis tools may be used to both identify and fix any issues. |
| 1. Architect and Design for Security Policies | Software should be designed such that security policies are implemented at all levels. To this end, security should be kept in mind during the architecture and design of the code. The need for applying differing privilege levels or separating into subsystems with differing privileges will be assessed and the option which provides the greatest security will be employed. |
| 1. Keep It Simple | Designs should be kept simple and concise to promote maintainability. Overly complex designs tend to have more errors when they are implemented, configured, and used. Complex solutions also require greater time and effort to maintain and ensure effectiveness. |
| 1. Default Deny | By default, any permission or access will be denied. Thus, users will need to be provided access rather than having access removed. Privileges will only be assigned as needed and will be removed when no longer needed. |
| 1. Adhere to the Principle of Least Privilege | Users or processes will only be given the access needed to perform their function. Permissions shall be elevated to what is necessary only for the period which they are needed to complete a task. In doing so, the potential for malicious actions being conducted with high privileges is reduced. |
| 1. Sanitize Data Sent to Other Systems | Prior to passing any data to other systems, it shall be validated and cleansed. Even if there is no malicious intent in sending data which could be harmful, there could still be something within which could cause a SQL injection attack, run a command, or attempt to attack another system. In sanitizing all data each time, the potential for a security or data breach is reduced. The responsibility to sanitize the data falls upon, but is not limited to, the process calling for data because the process is aware of the reason for calling for the data. |
| 1. Practice Defense in Depth | Multiple layers of defense shall be employed to minimize risk. Should one layer fail, another layer may prevent a flaw from exploitation or, if unable to prevent, reduce the harm caused. In layering defense, resource usage must be kept in mind as too many layers may lead to resources being exhausted. |
| 1. Use Effective Quality Assurance Techniques | Regular testing of the code and system must be conducted prior to any changes being pushed to prevent any issues, bugs, or security vulnerabilities. Additionally, tests should not be limited to just the segments changed as any changes implemented could cause unexpected issues or behavior in other parts of the code. Unit tests should be developed to provide through testing of the entire code. Other testing methods which can be employed are penetration testing, source code audits, and fuzz testing. Furthermore, in certain cases, independent third-party security review may be conducted to obtain a perspective free of bias. |
| 1. Adopt a Secure Coding Standard | Implementing and following standards for secure coding will help to minimize issues and vulnerabilities. It will increase efficiency as there will be fewer instances of needing to revisit and update code which does not follow the standards. Security should not be an afterthought in the development process but rather something that is incorporated in each stage of the development lifecycle. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | DCL-058-CPP | Do not modify the standard namespaces - Refrain from adding declarations or definitions to the standard std or posix namespaces, or any sub-namespaces therein. The sole exception to this rule is for template specializations that rely on user-defined types that fulfill the standard library's template requirements. |

| **Noncompliant Code** |
| --- |
| In this example, variable y is declared in the std namespace, which can cause undefined behavior. |
| **namespace** std {  **string** y;  } |

| **Compliant Code** |
| --- |
| In the corrected example, the variable y is placed in a namespace that is not reserved thus avoiding any conflicts. |
| **namespace** notstdspace {  **string** y;  } |

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

| **Principles(s):** The principle which applies to this standard is "**6. Adhere to the Principle of Least Privilege**". This principle states that the code should work with the least privileges needed to be able to function. Altering the standard namespace provides more access than is necessary thus going against this principle. This poses an issue because it can lead to undefined behavior, unintended side effects, and/or security flaws. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.STRUCT.DECL.SNM** | Modification of Standard Namespaces |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL58-a** | Do not modify the standard namespaces 'std' and 'posix' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: DCL58-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl58cpp.html) | Checks for modification of standard namespaces (rule fully covered) |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.32 | [**V1061**](https://pvs-studio.com/en/docs/warnings/v1061/) |  |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S3470**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-3470) |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | EXP-053-CPP | Do not read uninitialized memory - Objects with static or thread storage duration are automatically zero-initialized prior to any other form of initialization taking place, making explicit initialization unnecessary before accessing their values. |

| **Noncompliant Code** |
| --- |
| In this example a variable is initialized without being assigned a value and is subsequently printed which causes undefined behavior. |
| #include <iostream>    **void** y() {  **float** f;    std::cout << f;  } |

| **Compliant Code** |
| --- |
| In the corrected example, the variable f is initialized with a value of 0.00 before being used thus when it is printed it appropriately print '0.00'. |
| #include <iostream>    **void** y() {  **float** f = 0.00;    std::cout << f;  } |

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

| **Principles(s):** The principles which apply to this standard are "**4. Keep It Simple**" and "**10. Adopt a Secure Coding Standard**". Part of following a secure coding standard would include following best practices. A common best practice is to not read uninitialized memory to avoid unpredictable and unsafe behavior. Additionally, by ensuring that memory is always initialized before it gets used will simplify the behavior and allow you to avoid having to debug complex issues or worry about security issues related to uninitialized memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **uninitialized-read** | Partially checked |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -Wuninitialized clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.STRUCT.RPL LANG.MEM.UVAR** | Return pointer to local Uninitialized variable |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **53 D, 69 D, 631 S, 652 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-EXP53-a** | Avoid use before initialization |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: EXP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp53cpp.html) | Checks for:   * Non-initialized variable * Non-initialized pointer   Rule partially covered. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **uninitialized-read** | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR-053-CPP | Range check element access - Ensure that out-of-range values are not passed as arguments. Additionally, avoid calling functions which would access an index of a string, such as std::string::front()', on an empty string. |

| **Noncompliant Code** |
| --- |
| In this example, if the 'get\_index()' function returns a value greater than the range of the string 'f' then the code will attempt to access memory outside the bounds of 'f' which would lead to undefined behavior. |
| #include <string>    **extern** std::**size\_t** get\_index();    **void** q() {    std::string f("01234567");    f[get\_index()] = '5';  } |

| **Compliant Code** |
| --- |
| In the corrected example, the code is adjusted to check that the size of the string 'f' is greater than the index before setting attempting to access the index. If the index is not less than the size of string 'f' then something else will happen. Alternatively, exception handling may be used to catch the error and handle it. |
| #include <string>  extern std::size\_t get\_index();    **void** q() {  std::string f("01234567");  std::size\_t index = get\_index();    if (index < f.size()) {  s[index] = '1';  } else {  // Implement alternative or print message indicating error  }  } |

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

| **Principles(s):** The principles which apply to this standard are "**1. Validate Input Data**", "**4. Keep It Simple**", and "**5. Default Deny**". Range checking should be implemented to verify that an index exists before accessing an element. This prevents possible vulnerabilities or undefined behavior caused by out-of-bound access. By performing these checks regularly, the logic for string and array access becomes simpler and less likely to have errors. Furthermore, access to string elements, by default, should be denied until it has been confirmed that the element being access is within a valid range, thus preventing illegal access to memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR53-a** | Guarantee that container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: STR53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr53cpp.html) | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS-000-J | Prevent SQL injection - SQL injection can occur when a SQL query is left vulnerable to inputs which can be influenced by bad actors. If these attacked are not properly defended against, it can lead to a data breach in which the bad actor can steal or manipulate data. |

| **Noncompliant Code** |
| --- |
| In this example, the user provided inputs for 'username' and 'password' have not been sanitized, leaving it vulnerable to SQL injection. An attacker could use " ' OR '1'='1" for the username to have all rows in the username table returned. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  class SimpleLogin {  public Connection connect() throws SQLException {  String dbUrl = "jdbc:mysql://localhost:3306/db";  String username = "admin";  String password = "password123";  return DriverManager.getConnection(dbUrl, username, password);  }  public boolean auth(String username, String password) throws SQLException {  Connection conn = connect();  Statement stmt = conn.createStatement();  // SQL query vulnerable to SQL injection  String query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'";  ResultSet result = stmt.executeQuery(query);  boolean auth = result.next();  conn.close();  return auth;  }  } |

| **Compliant Code** |
| --- |
| In this example, prepared statements are utilized to ensure that the user's inputs are processed as parameters that are escaped automatically. This prevents SQL injection as the username and password inputs are handled as data rather than part of the SQL query. Additionally, the code logs any unsuccessful login attempts to identify potential attacks. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  class SecureLogin {  public Connection connect() throws SQLException {  String dbUrl = "jdbc:mysql://localhost:3306/db";  String username = "admin";  String password = "password123";  return DriverManager.getConnection(dbUrl, username, password);  }  public boolean authenticate(String username, String password) throws SQLException {  Connection conn = connect();  // Using a prepared statement to prevent SQL injection  String query = "SELECT \* FROM users WHERE username = ? AND password = ?";  PreparedStatement pstmt = conn.prepareStatement(query);  pstmt.setString(1, username);  pstmt.setString(2, password);  ResultSet result = pstmt.executeQuery();  boolean auth = result.next();  if (!auth) {  // Log unsuccessful attempt  logFailedAttempt(username);  }  conn.close();  return auth;  }  private void logFailedAttempt(String username) {  // Implementation for logging failed attempts  System.out.println("Failed login attempt for user: " + username);  }  } |

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

| **Principles(s):** The principles which apply to this standard are "**1. Validate Input Data**", "**3. Architect and Design for Security Policies**", and "**7. Sanitize Data Sent to Other Systems**". Any input should be validated to verify that it meets the requirement for type, format, and length. Additionally, the input and any input data should be sanitized to remove any characters that could be harmful. The possibility of security threats like SQL injection should be considered early on so the software can be designed with the necessary preventative measures, such as parameterized queries. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [The Checker Framework](https://wiki.sei.cmu.edu/confluence/display/java/The+Checker+Framework) | 2.1.3 | **Tainting Checker** | Trust and security errors (see Chapter 8) |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **JAVA.IO.INJ.SQL** | SQL Injection (Java) |
| [Findbugs](https://wiki.sei.cmu.edu/confluence/display/java/Findbugs) | 1.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| [Fortify](https://wiki.sei.cmu.edu/confluence/display/java/Fortify) | 1.0 | **HTTP\_Response\_Splitting** **SQL\_Injection\_\_Persistence** **SQL\_Injection** | Implemented |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/java/Klocwork) | 2024.2 | **SV.DATA.DB** **SV.SQL** **SV.SQL.DBSOURCE** | Implemented |
| [Parasoft Jtest](https://wiki.sei.cmu.edu/confluence/display/java/Parasoft) | 2024.1 | **CERT.IDS00.TDSQL** | Protect against SQL injection |
| [SpotBugs](https://wiki.sei.cmu.edu/confluence/display/java/SpotBugs) | 4.6.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** **SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-051-CPP | Properly deallocate dynamically allocated resources – If you do not deallocate memory properly, destructors can be prevented from being called which will lead to cause undefined behavior as memory was not properly deallocated. As such, memory needs to be deallocated in the same manner which it was allocated. |

| **Noncompliant Code** |
| --- |
| In this example, the object 'G' is constructed manually in the pre-allocated memory buffer using placement new. The object 'G' is then deleted using 'delete'. Utilized 'delete' attempts to free memory that wasn't allocated with new, which leads to undefined behavior. |
| #include <iostream>    struct G {  G() { std::cout << "G::G()" << std::endl; }  ~G() { std::cout << "G::~G()" << std::endl; }  };    void r() {  alignas(struct G) char buffer[sizeof(struct G)];  G \*g1 = new (&buffer) G;    // ...    delete g1;  } |

| **Compliant Code** |
| --- |
| In this example, the 'delete' call is removed and instead the destructor for 'g1' is called explicitly before the memory is released. |
| #include <iostream>    struct G {  G() { std::cout << "G::G()" << std::endl; }  ~G() { std::cout << "G::~G()" << std::endl; }  };    void r() {  alignas(struct G) char buffer[sizeof(struct G)];  G \*g1 = new (&space) G;    // ...    g1->~G();  } |

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

| **Principles(s):** The principles which apply to this standard are "**8. Practice Defense in Depth**", and "**10. Adopt a Secure Code Standard**". In adopting a secure code standard, there are best practices for coding which are also included. These best practice for managing memory minimize the risk of memory leaks, resource exhaustion, and undefined behavior. Properly deallocating the dynamically allocated memory also provides an additional layer of defense against these vulnerabilities, that could be utilized by bad actors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.FNH** **ALLOC.DF** **ALLOC.TM** **ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL-003-C | Use a static assertion to test the value of a constant expression – Assertions are used as a diagnostic tool to identify and eliminate any issues in code that could cause potential vulnerabilities which bad actors may attempt to exploit. |

| **Noncompliant Code** |
| --- |
| In this example the 'assert' statement is being used to confirm that the size of the struct 'counter' is the same as the combined size of the individual members ('unsigned char' and two instances of 'unsigned int') which it is comprised of. If the sizes do not match, the program crash when it is run which may indicate an issue with padding of the structure. |
| #include <assert.h>    struct counter {  unsigned char TYPE;  unsigned int VALUE;  unsigned int CURRENT;  };    int check\_size(void) {  assert(sizeof(struct counter) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| In this updated example, a check is implemented to confirm that the struct 'count' is the same size as the sum of it's parts. However, in this version of the code, if the sizes do not match, '#error' will be triggered and the compiler will return a messaging letting you know that the "Structure must not have any padding". |
| #include <assert.h>    struct counter {  unsigned char TYPE;  unsigned int VALUE;  unsigned int CURRENT;  };    #if (sizeof(struct counter) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** The principles which apply to this standard are "**2. Heed Compiler Warnings**", "**8. Practice Defense in Depth**", and "**10. Adopt a Secure Coding Standard**". Static assertions are checks at the time of compiling which can be considered like compiler warnings. They verify critical conditions are met to prevent defects that could lead to vulnerabilities. Performing these checks also adds an additional layer of defense by checking that the assumptions are true before the code executes. Static assertions allow developers to discover possible issues which results in a lower chance of vulnerabilities in the final product. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR-051-CPP | Handle all exceptions – Any exception thrown by the code must be handled with a matching handler. Exception handlers should be used even in situations which cannot be recovered from so as to appropriately unwind the stack and to allow for external resources to be managed properly before the process is terminated. |

| **Noncompliant Code** |
| --- |
| In this example 'dangerous\_task()' can throw exception when 'main()' calls it as a function of 'run()' but the exceptions are not caught because there is no matching handler which causes the program to terminate. |
| void dangerous\_task() noexcept(false);  void run() {  dangerous\_task();  }  int main() {  run();  } |

| **Compliant Code** |
| --- |
| In this updated example, a 'try-catch' block is implemented. The 'try' block is placed around the 'run()' function, so if 'dangerous\_task()' throws an exception when it is called by 'run()' the 'catch' block will catch and handle it. |
| void dangerous\_task() noexcept(false);  void run() {  dangerous\_task();  }  int main() {  try {  run();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** The principles which best apply to this standard are "**9. Use Effective Quality Assurance Techniques**", "**10. Adopt a Secure Coding Standard**". By catching all exceptions, errors are appropriately handled, and resources aren't exhausted thus possible data corruption or security breach are avoided. Proper exception handling is a best practice which would be included in the secure coding standards which are adopted. These best practices further minimize the likeliness of a security vulnerability related to unhandled exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Containers** | CTR-052-CPP | Guarantee that library functions do not overflow – When adding to the contents of a container, always ensure that there is enough space for the number of elements being added to prevent buffer overflow. When possible, the size of the container should be set so it matches the number of elements within it so no additional elements can be added maliciously or otherwise. |

| **Noncompliant Code** |
| --- |
| In this example, a vector consisting of integers is copied from 'input' to 'output'. However, 'output' is empty and the 'copy' function wont expand the size of the 'output' vector so buffer overflow occurs when the first element is attempted to be copied. |
| #include <algorithm>  #include <vector>    void process(const std::vector<int> &input) {  std::vector<int> output;  std::copy(input.begin(), input.end(), output.begin());  // ...  } |

| **Compliant Code** |
| --- |
| In this updated example, the size of 'output' has been expanded to match that of the size of 'input'. In doing this we ensure that 'output' is large enough to all of the elements of 'input'. Additionally, this helps prevent copying data that may be corrupt or tampered with. |
| #include <algorithm>  #include <vector>  void process(const std::vector<int> &input) {  // Initialize output with input.size() default-inserted elements  std::vector<int> output(input.size());  std::copy(input.begin(), input.end(), output.begin());  // ...  } |

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

| **Principles(s):** The principles which best apply to this standard are "**1. Validate Input Data**", "**4. Keep It Simple**", "**8. Practice Defense in Depth**". Any data being copied to a container should have its size validated prior to it being copied to prevent a buffer overflow which could lead to a security vulnerability or corrupted data. This simple step helps to prevent buffer overflow. However, check can be done at multiple layers to further prevent buffer overflow, in case one of the layers fails. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-CTR52-a** | Do not pass empty container iterators to std algorithms as destinations |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C++: CTR52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr52cpp.html) | Checks for library functions overflowing sequence container (rule partially covered). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Arrays** | ARR-039-C | Do not add or subtract a scaled integer to a pointer – Pointer math should be avoided where possible. However, in case where it must be used, it should only be used when the pointer is referring to an array, the array can consist of bytes. |

| **Noncompliant Code** |
| --- |
| In this example, the code tries to populate the array 'buffer' by using a pointer 'buffer\_ptr'. It utilizes a 'while' loop which is set to continue until 'buffer\_ptr' is less than 'buffer' + 'sizeof(buffer)'. However, this can cause out of bounds memory access to occur as 'sizeof(buffer)' will return the size in bytes. |
| enum { BUFFER\_SIZE = 80 };    extern int retrieve\_data(void);  int buffer[BUFFER\_SIZE];    void execute(void) {  int \*buffer\_ptr = buffer;    while (buffer\_ptr < (buffer + sizeof(buffer))) {  \*buffer\_ptr++ = retrieve\_data();  }  } |

| **Compliant Code** |
| --- |
| In the corrected version of the example, the 'while' loop is updated to continue as long as 'buf\_ptr' is less than the value of 'buffer' + 'BUFFER\_SIZE' which will make sure that our of bounds memory is not accessed. |
| enum { BUFFER\_SIZE = 80 };    extern int retrieve\_data(void);  int buffer[BUFFER\_SIZE];    void execute(void) {  int \*buffer\_ptr = buffer;    while (buffer\_ptr < (buffer + BUFFER\_SIZE)) {  \*buffer\_ptr++ = retrieve\_data();  }  } |

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

| **Principles(s):** The principles which best apply to this standard are "**1. Validate Input Data**", "**2. Keep It Simple**", and "**8. Practice Defense in Depth**". Keeping code simple by not including any complex pointer arithmetic will reduce the probability of creating errors that can cause undefined behavior or expose vulnerabilities. If arithmetic is done using pointers, verify that the pointer is referencing a valid array to prevent memory access violations or buffer overflows. Additionally, performing these checks and validating the information can provide an additional layer to identify possible errors which other areas of the code might not. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **scaled-pointer-arithmetic** | Partially checked  Besides direct rule violations, Astrée reports all (resulting) out-of-bound array accesses. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-ARR39** | Fully implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** **LANG.STRUCT.PARITH** **LANG.STRUCT.PBB** **LANG.STRUCT.PPE** **BADFUNC.BO.\*** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun Pointer Arithmetic Pointer before beginning of object Pointer past end of object A collection of warning classes that report uses of library functions prone to internal buffer overflows. |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | **scaled-pointer-arithmetic** | Partially checked  Besides direct rule violations, Astrée reports all (resulting) out-of-bound array accesses. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Output (FIO)** | FIO-051-CPP | Close files when they are no longer needed – Whenever a file is opened, it must also be closed. This should be done when the file is no longer needed or before the program ends, whichever of these two occurs sooner. |

| **Noncompliant Code** |
| --- |
| In this example, the code attempts to open the file 'filePath'. If it is unable to open the file an error is returned. Once the file has been opened, the code terminates. The code does not call a destructor or close the file, it instead call 'terminate()' resulting in the file not being closed properly. |
| #include <exception>  #include <fstream>  #include <string>    void handle\_file(const std::string &filePath) {  std::fstream dataStream(filePath);  if (!dataStream.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this correct example, a simple call to close the file is made, 'dataStream.close()'. The code also performs a check to verify the file properly closed. If the file did not close properly, the code will do something to handle it. Otherwise the process is terminated. |
| #include <exception>  #include <fstream>  #include <string>    void handle\_file(const std::string &filePath) {  std::fstream dataStream(filePath);  if (!dataStream.is\_open()) {  // Handle error  return;  }  // ...  dataStream.close();  if (dataStream.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):** The principles which best apply to this standard are "**6. Adhere to the Principle of Least Privilege**" and "**8. Practice Defense in Depth".** Files should be closed immediately after they are not going necessary. This will reduce exposure to private information and minimize the possibility of unauthorized access which adheres to the principle of least privilege. Being consistent in closing files also reduces the chance of exhausting resources and possible breaches of security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) |  |  | Runtime detection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: FIO51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio51cpp.html) | Checks for resource leak (rule partially covered) |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.LEAK** | Leak |

### 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 will be integrated throughout the DevSecOps lifecycle to enforce compliance with the security standards defined in this policy. Green Pace’s existing DevOps process will be enhanced by introducing automated tools for code analysis, security testing, and continuous monitoring. This transition will evolve the process into a more secure and robust DevSecOps lifecycle, ensuring that security is consistently prioritized at every stage of development.

In the Pre-Production Phase, automated code analysis tools, such as SonarQube and CodeSonar, will be embedded into the CI/CD pipeline. These tools will continuously scan the codebase for compliance with secure coding standards, including checks for memory safety, input validation, and SQL injection vulnerabilities. Additionally, automated security testing, including penetration testing with tools like OWASP ZAP, will be systematically scheduled within the CI/CD pipeline. This approach will identify and address any security flaws before the code is deployed, ensuring that the software is secure from the outset.

During the Production Phase, automated deployment and configuration tools, such as Ansible and Terraform, will be utilized to enforce security configurations. These tools will ensure that only compliant builds are deployed and that essential security settings, such as encryption in transit and at rest, are automatically applied. Continuous monitoring will be implemented using Security Information and Event Management (SIEM) tools, which will monitor system logs, firewall logs, and application logs for any anomalies. Tools like Splunk and ELK Stack will provide real-time alerts and enable automated responses to detected threats, maintaining a high level of security during production.

In the Post-Production Phase, automated incident response tools will be employed to swiftly address detected threats. For example, if an unauthorized access attempt is identified, the system can automatically revoke access and re-encrypt data, thereby preventing any potential breaches. Further investigation will then be conducted to assess the breach and implement additional safeguards to fortify the system against future threats.

Furthermore, a neural network model, similar to AlphaZero, will be integrated into the security framework to continually learn and adapt to new attack patterns. This model will be trained on existing security policies and will simulate attacks to identify potential vulnerabilities. Over time, the model will enhance its defense mechanisms, helping to preemptively address threats before they can be exploited, ensuring that Green Pace remains ahead of emerging security challenges.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL-058-CPP | High | Unlikely | Medium | P6 | L2 |
| EXP-053-CPP | High | Probable | Medium | P12 | L1 |
| STR-053-CPP | High | Unlikely | Medium | P6 | L2 |
| IDS-000-J | High | Likely | Medium | P18 | L1 |
| MEM-051-CPP | High | Likely | Medium | P18 | L1 |
| DCL-003-C | Low | Unlikely | High | P1 | L3 |
| ERR-051-CPP | Low | Probable | Medium | P4 | L3 |
| CTR-052-CPP | High | Likely | Medium | P18 | L1 |
| ARR-039-C | High | Probable | High | P6 | L2 |
| FIO-051-CPP | Medium | Unlikely | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the protection of data on the device or in a database, even when the storage itself is no longer reliable or trusted. The purpose of this policy is to ensure that our data has another layer of defense, in the case that they are stolen. This is done by encrypting and decrypting data through the utilization of a key. If the user does not have the key, they will not be able to decrypt the data. A bad actor could attempt to brute force the encryption, but this could take several years and not be an effective method for the bad actor to access the data. Implementing measures like this disincentivizes bad actors from attempting to steal the data. Any sensitive or confidential data should be encrypted. |
| Encryption in flight | Encryption in flight refers to encrypting data which is being transferred over a network. This ensures that bad actors who may manage to steal the data when it is mid transfer are still not able to easily access it. Transport layer security (TLS) is utilized to secure the data while it is being transferred. This protocol encrypts the data prior to sending it, authenticates the sender and recipients, and verifies that the transferred data has not been tampered with. |
| Encryption in use | Encryption in use refers to the protection of data while it is being processed or in memory. It prevents programs or users which have not been authorized from accessing the data. This can be achieved by utilizing protected memory or homomorphic encryption. Homomorphic encryption allows for encrypted data to be manipulated even without having the necessary key to decrypt. By utilizing these types of protections, we minimize the risk of attacks on application server. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process by which a user's identity is confirmed. Multi-factor authentication (MFA) has become the standard now. MFA utilizes multiple authentication checks by requesting the user's password before, typically, sending a prompt to the user's cellphone to confirm it is them attempting to log in. After confirming they are attempting to login, the user is prompted to verify that they are the owner of the device by using a biometric scan or entering the device's unlock password. This level of verification lets us be certain the person trying to access the system or data is truly who they say they are. All new users will need to set up these systems on their device and utilize them when they attempt to login. |
| Authorization | After a user has been authenticated, a check still needs to be conducted to verify they are authorized to access the system or data. This check is done every time a user attempts to access a system or data. The administrator for the system should be auditing and updating the user-based access controls regularly to ensure that the appropriate people have access. In addition to auditing and updating existing users, new users must also be added and set with the appropriate access controls. |
| Accounting | Even after a user has been verified to be who they are and to have access to the data or system, all requests and changes made by the user are logged. This information provides an audit trail to determine what systems or data may have been accessed, modified, or added by a bad actor in the event of a breach. This information can also be used to track patterns of access to identify and report abnormalities. |

**\***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 |  |
| 1.5 | 07/21/2024 | 1. Provided explanation for each of the 10 principles.  2. Labeled, named, and rationalized the 10 selected coding standards.  3. Added and explained examples of compliant and non-compliant code blocks for each of the 10 coding standards selected | Sheel Patel | Toni Farley |
| 2.0 | 08/11/2024 | 1. Updated explanations of the principles, coding standards, code examples, and explanations per feedback received.  2. Added risk assessment for each coding standard  3. Determined tool or tools to use for automated detection of issues  4. Provided explanation of where in the DevSecOps lifecycle automation should take place.  5. Updated the summarized risk assessment table with the information from the 10 coding standards selected  6. Developed policies for encryption in flight, at rest, and in use and for the three elements of the Triple-A framework.  7. Mapped the applicable principles of the 10 core security principles to each of the 10 coding standards and provided explanation as to how they apply. | Sheel Patel | Toni Farley |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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