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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data is crucial to protecting the security of the application, since it can prevent security attacks such as SQL injections. For instance, users with malicious intent, can insert SQL code into the program’s input field, in order to gain access to a restricted database. By implementing input validation, the program can prevent SQL injections by ensuring only acceptable inputs are used. |
| 1. Heed Compiler Warnings | Compilers are typically equipped with warning messages, informing the developer of certain logical and runtime errors. There are some errors detected within the program, that cannot prevent the program from running. However, they might create security vulnerabilities within the program, which result in future security attacks. |
| 1. Architect and Design for Security Policies | According to the Educause article, “Security Architecture and Design” (2024), architect and design for security policies is defined as: “Security architecture and design looks at how information security controls and safeguards are implemented in IT systems in order to protect the confidentiality, integrity, and availability of the data that are used, processed, and stored in those systems.” (Educause, 2024). This requires intentional thought and purpose when designing a security policy. |
| 1. Keep It Simple | When designing a security policy, it is best to keep simplicity in mind. This is important, as it is easy to dismiss the program’s actual security needs when tackling multiple items at the same time. |
| 1. Default Deny | The NIST Computer Security Resource Center glossary section, “Deny by Default” (2024), defines default deny with the following definiton: “To block all inbound and outbound traffic that has not been expressly permitted by firewall policy.” (NIST Computer Security Resource Center, 2024). It is imperative for the developer to have a solid firewall policy that will meet the security needs of the program. |
| 1. Adhere to the Principle of Least Privilege | Alexander S. Gillis in the TechTarget article, “What is the principle of least privilege?” (2023), gives the following definition to the principle of least privilege: “The principle of least privilege (POLP) is a concept in computer security that limits users' access rights to only what is strictly required to do their jobs.” (Gillis, 2023). This will require complete knowledge to the required storage needs, in order for the administrator to determine which users gets the appropriate security access. |
| 1. Sanitize Data Sent to Other Systems | The TechTarget article, “Data sanitization techniques: Standards, practices, legislation” by Paul Kirvan (2020), makes the following point about data sanitization: “An effective data sanitization process lessens the chance that your organization's valuable data could be stolen or compromised, and enhances compliance.” (Kirvan, 2020). Incorporating data sanitization will require a thorough evaluation of the security needs of the program, in order to determine which data can be considered obsolete for sanitization. |
| 1. Practice Defense in Depth | According to the Fortinet article, “What Is Defense In Depth?” (2024), defense in depth is: “[...]a strategy that leverages multiple security measures to protect an organization's assets.” (Fortinet, 2024). In other words, more than one security solution is used in conjunction, in order for the best security policy possible. |
| 1. Use Effective Quality Assurance Techniques | Kristi Johnson in the Yoh article, “Security quality assurance: Exploring the roles and responsibilities of this growing Field” (2023), defines security quality assurance as: “[...]the process of ensuring that software and systems meet established security requirements and standards.” (Johnson, 2023). This will typically be done at the end of the phase, in order to ensure that the program has met all of the security requirements. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves meticulous planning and researching the best possible coding standard, that will the best fit for the program’s needs and requirements. This will result in a secure program. |

### 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** | [EXP58-CPP.] | Pass an object of the correct type to va\_start |

**Rationalize the Standard:**

This standard will be used to ensure that the program will obtain proper data type, in order to prevent erros

such as undefined behaviors.

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the object passed to va\_start() will undergo a default argument promotion, which results in undefined behavior. |
| #include <cstdarg>  extern "C" void f(float a, ...) {  va\_list list;  va\_start(list, a);  // ...  va\_end(list);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f() accepts a double instead of a float. |
| #include <cstdarg>  extern "C" void f(double a, ...) {    va\_list list;    va\_start(list, a);    // ...    va\_end(list);  } |

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

| **Principles(s):** Validate input data, Heed Compiler Warnings, Default deny  - Coding standard #1 maps to principle #1, “Validate input data”, since it involves utilizing logic to ensure that only the appropriate data type is used. This will prevent errors from occuring.  - Coding standard #2 is also applicable, since the compiler will usually generate an error message whenever an incorrect data type has been returned.  - Coding standard #4 is relevant, since this standard can be easily implemented, through the use of conditional statements and exception-handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT C++: EXP58-CPP | Checks for incorrect data types for second argument of va\_start (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [MSC-52-CPP] | Value-returning functions must return a value from all exit paths |

**Rationalize the Standard:**

This standard will be used to ensure that the program return the correct input value.

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| int absolute\_value(int a) {    if (a < 0) {      return -a;    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| int absolute\_value(int a) {    if (a < 0) {      return -a;    }    return a;  } |

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

| **Principles(s): 1.** Validate Input Data, Heed Compiler Warnings  - Security #1 is chosen, since it involves validating the input, to ensure only the correct data type is returned.  - Coding standard #2 is also applicable, since the compiler will usually generate an error message whenever an incorrect data type has been returned. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT C++: MSC52-CPP | Checks for missing return statements (rule partially covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [FIO-47-C. ] | Use valid format strings |

**Rationalize the Standard:**

This standard is used to ensure that string variables have a valid format, in order to prevent errors related to conversion.

| **Noncompliant Code** |
| --- |
| Mismatches between arguments and conversion specifications may result in undefined behavior. Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the error\_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. Likewise, the error\_msg argument is incorrectly matched with the d specifier instead of the s specifier. These usages result in undefined behavior. One possible result of this invocation is that printf() will interpret the error\_type argument as a pointer and try to read a string from the address that error\_type contains, possibly resulting in an access violation. |
| #include <stdio.h>  void func(void) {    const char \*error\_msg = "Resource not available to user.";    int error\_type = 3;    /\* ... \*/    printf("Error (type %s): %d\n", error\_type, error\_msg);    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications: |
| #include <stdio.h>  void func(void) {    const char \*error\_msg = "Resource not available to user.";    int error\_type = 3;    /\* ... \*/    printf("Error (type %d): %s\n", error\_type, error\_msg);    /\* ... \*/  } |

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

| **Principles(s):** Validate input data, Sanitize Data Sent to Other Systems, Heed Compiler Warnings  - Principle #1 will be used, since the string should be of the proper format, before being used as an input.  - Principle #7 is chosen, since it will ensure that only strings of the proper format is used.  - Coding standard #2 is applicable, since the compiler will usually generate an error message whenever a string is in an incorrect format. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT C: Rule FIO47-C | Check for format string specifiers and arguments mismatch (rule fully covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [IDS-00-J. ] | Prevent SQL injection |

**Rationalize the Standard:**

This standard is sufficient for preventing SQL injections by implementing prepared statements.

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a char array, the database connection is created, and then the passwords are hashed.  Unfortunately, this code example permits a SQL injection attack by incorporating the unsanitized input argument username into the SQL command, allowing an attacker to inject validuser' OR '1'='1. The password argument cannot be used to attack this program because it is passed to the hashPassword() function, which also sanitizes the input. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  class Login {    public Connection getConnection() throws SQLException {      DriverManager.registerDriver(new              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"      return DriverManager.getConnection(dbConnection);    }    String hashPassword(char[] password) {      // Create hash of password    }    public void doPrivilegedAction(String username, char[] password)                                   throws SQLException {      Connection connection = getConnection();      if (connection == null) {        // Handle error      }      try {        String pwd = hashPassword(password);        String sqlString = "SELECT \* FROM db\_user WHERE username = '"                           + username +                           "' AND password = '" + pwd + "'";        Statement stmt = connection.createStatement();        ResultSet rs = stmt.executeQuery(sqlString);        if (!rs.next()) {          throw new SecurityException(            "User name or password incorrect"          );        }        // Authenticated; proceed      } finally {        try {          connection.close();        } catch (SQLException x) {          // Forward to handler        }      }    }  } |

| **Compliant Code** |
| --- |
| The JDBC library provides an API for building SQL commands that sanitize untrusted data. The java.sql.PreparedStatement class properly escapes input strings, preventing SQL injection when used correctly. This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument 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        }      }    }  } |

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

| **Principles(s):** #1. Validate Input Data, Architect and Design for Security Policies  - Principle #1 is chosen, since it input validation will prevent SQL injection attacks from happening. It will accomplish this by preventing the program from accepting SQL commands as input values.  - Principle #3 will be used, since every security policy at a minimum, should account for SQL injections. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft Jtest | 2024.1 | CERT.IDS00.TDSQL | Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-30-C ] | Do not access freed memory |

**Rationalize the Standard:**

This standard is used to ensure that the proper reference is used in order to protect the memory.

| **Noncompliant Code** |
| --- |
| This example from Brian Kernighan and Dennis Ritchie [Kernighan 1988] shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, p is freed before p->next is executed, so that p->next reads memory that has already been freed. |
| #include <stdlib.h>  struct node {    int value;    struct node \*next;  };  void free\_list(struct node \*head) {    for (struct node \*p = head; p != NULL; p = p->next) {      free(p);    }  } |

| **Compliant Code** |
| --- |
| Kernighan and Ritchie correct this error by storing a reference to p->next  in q before freeing p: |
| #include <stdlib.h>  struct node {    int value;    struct node \*next;  };  void free\_list(struct node \*head) {    struct node \*q;    for (struct node \*p = head; p != NULL; p = q) {      q = p->next;      free(p);    }  } |

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

| Principles(s): #6. Adhere to the Principle of Least Privilege, #7 Sanitize Data Sent to Other Systems  - Principle #6 is chosen, since it will limit the access of data, depending on the amount of privileges granted.  - Principe #7 will be used, since a variable must be tested to see if it is free, before being used. |
| --- |

**Threat Level**

| **SeveritySanSan**itize Data Sent to Other Systems | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses after its deallocation and double memory deallocations |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [DCL-03-C] | Use a static assertion to test the value of a constant expression |

**Rationalize the Standard:**

This standard is used to handle errors, to determine correct code.

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #include <assert.h>  struct timer {    unsigned char MODE;    unsigned int DATA;    unsigned int COUNT;  };  int func(void) {    assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| struct timer {    unsigned char MODE;    unsigned int DATA;    unsigned int COUNT;  };  #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** #4 Keep it simple, 9. Use Effective Quality Assurance Techniques  - Principle #4 will be used, since assertion statements should be kept simple.  - Principle #9 will be used, due to the fact that it will allow for a series of tests to be conducted, such as exception testing. This will be done on a constant expression. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [ERR-56-CPP] | Guarantee exception safety |

**Rationalize the Standard:**

This standard is used to handle errors, in order to prevent program failure.

| **Noncompliant Code** |
| --- |
| The following noncompliant code example shows a flawed copy assignment operator. The implicit invariants of the class are that the array member is a valid (possibly null) pointer and that the nElems member stores the number of elements in the array pointed to by array. The function deallocates array and assigns the element counter, nElems, before allocating a new block of memory for the copy. As a result, if the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior. |
| #include <cstring>  class IntArray {    int \*array;    std::size\_t nElems;  public:    // ...    ~IntArray() {      delete[] array;    }    IntArray(const IntArray& that); // nontrivial copy constructor    IntArray& operator=(const IntArray &rhs) {      if (this != &rhs) {        delete[] array;        array = nullptr;        nElems = rhs.nElems;        if (nElems) {          array = new int[nElems];          std::memcpy(array, rhs.array, nElems \* sizeof(\*array));        }      }      return \*this;    }    // ...  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the copy assignment operator provides the strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. Only after the allocation succeeds does the function proceed to change the state of the object. In addition, by copying the array to the newly allocated storage before deallocating the existing array, the function avoids the test for self-assignment, which improves the performance of the code in the common case [Sutter 2004]. |
| #include <cstring>  class IntArray {    int \*array;    std::size\_t nElems;  public:    // ...    ~IntArray() {      delete[] array;    }    IntArray(const IntArray& that); // nontrivial copy constructor    IntArray& operator=(const IntArray &rhs) {      int \*tmp = nullptr;      if (rhs.nElems) {        tmp = new int[rhs.nElems];        std::memcpy(tmp, rhs.array, rhs.nElems \* sizeof(\*array));      }      delete[] array;      array = tmp;      nElems = rhs.nElems;      return \*this;    }    // ...  }; |

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

| **Principles(s):** #9. Use Effective Quality Assurance Techniques  - Principle #9 will be used, since it will ensure that exception testing is properly guarded. This will prevent failures when testing for exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions  Do not leave 'catch' blocks empty |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Correct Integer Type | [INT-35-C] | Use correct integer precisions |

**Rationalize the Standard:**

This standard is used to ensure that the correct integer is used in order to prevent input error.

| **Noncompliant Code** |
| --- |
| This noncompliant code example illustrates a function that produces 2 raised to the power of the function argument. To prevent undefined behavior in compliance with INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand, the function ensures that the argument is less than the number of bits used to store a value of type unsigned int. |
| #include <limits.h>  unsigned int pow2(unsigned int exp) {    if (exp >= sizeof(unsigned int) \* CHAR\_BIT) {      /\* Handle error \*/    }    return 1 << exp;  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a popcount() function, which counts the number of bits set on any unsigned integer, allowing this code to determine the precision of any integer type, signed or unsigned. |
| #include <stddef.h>  #include <stdint.h>  /\* Returns the number of set bits \*/  size\_t popcount(uintmax\_t num) {    size\_t precision = 0;    while (num != 0) {      if (num % 2 == 1) {        precision++;      }      num >>= 1;    }    return precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s): #1.** Validate input data, #2 Heed compiler warnings  - Principle #1 will ensure that only the correct integer precision is used. This will prevent input errors from occuring.  - Since the compiler will typically generate a warning when an integer has an incorrect precision, principle #2 should be used since the warnings can prevent this issue. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT35-a | Use correct integer precisions when checking the right hand operand of the shift operator |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Arrays | [ARR-36-C] | Do not subtract or compare two pointers that do not refer to the same array |

**Rationalize the Standard:**

This standard will be used to prevent errors between two pointers.

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, pointer subtraction is used to determine how many free elements are left in the nums array: |
| #include <stddef.h>  enum { SIZE = 32 };  void func(void) {    int nums[SIZE];    int end;    int \*next\_num\_ptr = nums;    size\_t free\_elements;    /\* Increment next\_num\_ptr as array fills \*/    free\_elements = &end - next\_num\_ptr;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the number of free elements is computed by subtracting next\_num\_ptr from the address of the pointer past the nums array. While this pointer may not be dereferenced, it may be used in pointer arithmetic. |
| #include <stddef.h>  enum { SIZE = 32 };  void func(void) {  int nums[SIZE];  int \*next\_num\_ptr = nums;  size\_t free\_elements;  /\* Increment next\_num\_ptr as array fills \*/  free\_elements = &(nums[SIZE]) - next\_num\_ptr;  } |

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

| **Principles(s):** #7. Sanitize Data Sent to Other Systems  Principle #7 is used, in order to ensure that before two pointers are used, they do not refer to the same array. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT C: Rule ARR36-C | Checks for subtraction or comparison between pointers to different arrays (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output (Closed File) | [FIO-46-C] | Do not access a closed file |

**Rationalize the Standard:**

This standard is used to prevent closed files from being opened.

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the stdout stream is used after it is closed: |
| #include <stdio.h>  int close\_stdout(void) {    if (fclose(stdout) == EOF) {      return -1;    }    printf("stdout successfully closed.\n");    return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, stdout is not used again after it is closed. This must remain true for the remainder of the program, or stdout must be assigned the address of an open file object. |
| #include <stdio.h>  int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }  fputs("stdout successfully closed.", stderr);  return 0;  } |

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

| **Principles(s):** #6. Adhere to the Principle of Least Privilege  Principle #6 will be used, since it will ensure that only users with the proper privileges, are allowed access. This will prevent them from accessing closed files. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_C-FIO46-a | Do not use resources that have been freed |

### 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.

### (done) 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.

### (done) 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.

### (done) 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.

### (done) 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.

**PROMPT: Summarize, in general, how automation (tools) will be used for the enforcement of and compliance to the standards defined in this policy.**

Automation is a very important tool, that can be utilized in any phase of the software development life cycle (SDLC). The tool can also be used to satisfy the security needs of any application. In our context, automation would greatly improve the workflow of Green Pace’s DevOps process. Within the established DevOps process, there are eight steps: 1) Assess and plan, 2) Design, 3) Build, 4) Verify and test, 5) Transition and health check, 6) Monitor and detect, 7) Respond, and 8) Maintain and stabilize. All of this is found in the provided image shown above. With all of this in mind, automation would be very effective in the fourth phase of the DevOps process, which pertains to the verification and testing of the program.

Testing is a process that should be done in almost every step of an application’s life cycle. This is especially true in the phase of the application, where it is about to be shipped to the public. Adequate testing is required, or else the application runs the risk of being shipped with underlying vulnerabilities and bugs. The developer can ensure that the program will meet all of the security requirement before shipment, by utilizing automation in this phase. This can be done through the use of unit tests. Unit tests allows the developer to test the program in a number of use cases, in order to test for its performance. If done manually, the process can take a long time. This can result in some crucial areas being overlooked. By having an automated testing system, the developer can verify that the program meets all of its requirements, by testing for every possible unit case. The importance of automation tools can be found in the tenth coding standard: “[ARR-36-C]: Do not subtract or compare two pointers that do not refer to the same array”. To meet this standard, the developer can utilize the Polyspace Bug Finder, which will prevent two pointers from making comparisons or subtractions if they refer to the same array.

### (done) Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| [EXP58-CPP.] | Medium | Unlikely | Medium | P4 | L3 |
| [MSC-52-CPP] | Medium | Probable | Medium | P8 | L2 |
| [FIO-47-C] | High | Unlikely | Medium | P6 | L2 |
| [IDS-00-J. | High | Likely | Medium | P18 | L1 |
| [MEM-30-C ] | High | Likely | Medium | **P18** | L1 |
| [DCL-03-C] | Low | Unlikely | High | P1 | L3 |
| [ERR-56-CPP] | High | Likely | High | P9 | L2 |
| [INT-35-C] | Low | Unlikely | Medium | P2 | L3 |
| [ARR-36-C] | Medium | Probable | Medium | P8 | L2 |
| [FIO-46-C] | MEDIUM | Unlikely | Medium | P4 | L3 |

### (here) 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 | According to Sabrina Lupșan in the *Cyscale* article, “Types of Encryption for in Motion, in Use, at Rest Data” (2022), encryption at rest involves data that has arrived at its destination, but is not used (Lupșan, 2022). This policy can be applied by adding an encryption algorithm to encrypt data that has been received, but not used. Although the data is not used, its contents should still be encrypted, due to the risk of it being accessed due to a data breach. |
| Encryption in flight | Sabrina Lupșan, in the previously mentioned *Cyscale* article (2022), describes encryption in motion (flight), as involving data that is moved from different locations (Lupșan, 2022). Encryption in flight can be implemented by encrypting data, regardless of its destination. It is possible for security attackers to intercept data when it is in transit. Therefore, every data that is traveling should be encrypted. |
| Encryption in use | Lupșan, in the same *Cyscale* article, writes that encryption in use deals with data that is accessed and used (Lupșan, 2022). Encryption in use will be implemented by encrypting data that is used after arriving successfully at its destination. This will prevent data from being accessed by unauthorized users. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | According to the *Fortinet* article, “What Is Authentication, Authorization, And Accounting (AAA) Security?” (2024), the first part of the Triple-A framework, authentication, involves: “[…] a user providing information about who they are.” (Fortinet, 2024). Authentication can be implemented by providing every user with a token, which will serve as an identifier. This is needed in order to provide the users with a proof of identity, which will prevent impostors from accessing the owner’s account. Authentication is usually done whenever a new user has been created. |
| Authorization | The previously mentioned *Fortinet* article (2024), describes authorization in the following: “[d]uring authorization, a user can be granted privileges to access certain areas of a network or system.” (Fortinet, 2024). This second part of the Triple-A framework will be implemented, by granting users with certain privileges. By doing so, unauthorized activity within the program is prevented. |
| Accounting | The same Fortinet article (2024) writes that accounting involves keeping track of every user activity (Fortinet, 2024). Accounting can be implemented by having the program keep a log of a user’s every activity, whether they have logged in, or have accessed or made changes to a particular database. By accounting for every time a user has accessed the system, the system administrator is able to monitor for any suspicious activity, as well as make a determination on an attacker’s identity. |

**\***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

### (done) 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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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