**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 | This principle emphasizes the importance of thoroughly validating all input data received by the software. By doing so, potential vulnerabilities such as SQL injection, cross-site scripting (XSS), and buffer overflows can be mitigated. |
| 1. Heed Compiler Warnings | Developers should pay close attention to compiler warnings and errors. Ignoring or suppressing warnings can lead to security issues, as they may indicate potential vulnerabilities or coding mistakes that need to be addressed. |
| 1. Architect and Design for Security Policies | Security should be integrated into the software's architecture and design from the outset. This principle highlights the importance of considering security requirements and policies during the design phase to prevent vulnerabilities from being introduced later. |
| 1. Keep It Simple | Simplicity in software design and implementation is a key principle for security. Complex code is often harder to review and can introduce unnecessary vulnerabilities. Keeping code simple helps reduce the attack surface and makes it easier to maintain and secure. |
| 1. Default Deny | The default stance for software should be to deny access or permissions. This means that unless explicitly allowed, all access should be denied. This principle minimizes the risk of unauthorized access and helps enforce the principle of least privilege. |
| 1. Adhere to the Principle of Least Privilege | Users, processes, and systems should only have the minimum level of access or permissions necessary to perform their required tasks. This reduces the potential impact of security breaches and limits the exposure of sensitive data. |
| 1. Sanitize Data Sent to Other Systems | Data sent to external systems or components should be sanitized and validated to prevent malicious data from being used to exploit vulnerabilities in those systems. Failure to do so could lead to security breaches. |
| 1. Practice Defense in Depth | This principle promotes a multi-layered security approach. Instead of relying solely on one security measure, multiple layers of security controls are implemented. This includes firewalls, intrusion detection systems, access controls, and more to create a stronger defense against threats. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance should include security testing and validation. Automated and manual security testing, code reviews, and penetration testing are examples of techniques used to identify and remediate security vulnerabilities before deployment. |
| 1. Adopt a Secure Coding Standard | Developers should follow a secure coding standard or guidelines specific to the programming language being used. These standards provide best practices and rules for writing secure code, helping to prevent common coding mistakes that can lead to vulnerabilities. |

### 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** | DCL05-C | Use typedrfs of non-pointer types only |

| **Noncompliant Code** |
| --- |
| The following type definition improves readability at the expense of introducing a const-correctness issue. In this example, the const qualifier applies to the typedef instead of to the underlying object type. Consequently, func does not take a pointer to a const struct obj but instead takes a const pointer to a struct obj. |
| struct obj {  **int** i;  **float** f;  };  typedef struct obj \*ObjectPtr;    void func(const ObjectPtr o) {    /\* Can actually modify o's contents, against expectations \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution makes use of type definitions but does not declare a pointer type and so cannot be used in a const-incorrect manner: |
| struct obj {  **int** i;  **float** f;  };  typedef struct obj Object;    void func(const Object \*o) {    /\* Cannot modify o's contents \*/  } |

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

| **Principles(s):** This coding standard uses the concept "DCL05-C," which stands for "Declare objects with appropriate storage durations." This idea is covered in the CERT C Secure Coding Standard, which offers recommendations for producing trustworthy and secure C code.  Description of how DCL05-C corresponds to this requirement:  Developers are aided in declaring objects with the proper storage duration and type by DCL05-C. This standard places a strong emphasis on using type definitions appropriately and maintaining const-correctness by applying the const qualifier to the correct level of the type hierarchy in order to prevent unexpected code behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| low | unlikely | medium | P2 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **pointer-typedef** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL05** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **LANG.STRUCT.PIT** | Pointer type inside typedef |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **299 S** | Partially implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT31-C | Ensure that integer conversions do not result in lost or misinterpreted data |

| **Noncompliant Code** |
| --- |
| Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code example results in a truncation error on most [implementations](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-implementation): |
| #include <limits.h>    void func(void) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    sc = (**signed** **char**)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type: |
| #include <limits.h>    void func(void) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    if (u\_a <= SCHAR\_MAX) {      sc = (**signed** **char**)u\_a;  /\* Cast eliminates warning \*/    } else {      /\* Handle error \*/    }  } |

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

| **Principles(s):** This coding standard's guiding principle is referred to as "Lossless Data Conversion." This principle highlights the need to take care when converting data between data types, especially when doing so from a wider type (such as an unsigned integer) to a smaller type (such as a signed integer), to prevent data loss or misinterpretation. To avoid truncation, overflow, or sign mistakes, in other words, throughout the conversion process.  The concept of lossless data conversion maps as follows in the context of the offered coding standard "INT31-C," which explicitly addresses integer conversions:  Validate Data Range: The sample code that complies with the standard shows how to put the idea into practice. It determines whether the signed char data type's (SCHAR\_MAX) range can accommodate the value of the unsigned integer variable u\_a. This step makes sure that there will be no data loss via truncation as a result of the conversion.  Error Handling: The complying code handles an error appropriately if the validation check finds that the conversion would cause data loss (i.e., u\_a is higher than SCHAR\_MAX). Because performing an erroneous conversion could result in unpredictable and possibly inaccurate behavior in the application, error management is essential.  The complying code follows the notion of lossless data conversion since it works to avoid data loss or misinterpretation when converting from an unsigned integer to a signed char by checking the data range before executing the conversion and managing potential problems. This strategy encourages the quality and safety of the code, lowering the possibility of undesired behavior brought on by bad data conversions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **signed\_downcast** | Exhaustively verified. |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **memsetValueOutOfRange** | The second argument to memset() cannot be represented as unsigned char |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.26 | [**V562**](https://pvs-studio.com/en/docs/warnings/v562/), [**V569**](https://pvs-studio.com/en/docs/warnings/v569/), [**V642**](https://pvs-studio.com/en/docs/warnings/v642/), [**V676**](https://pvs-studio.com/en/docs/warnings/v676/), [**V716**](https://pvs-studio.com/en/docs/warnings/v716/), [**V721**](https://pvs-studio.com/en/docs/warnings/v721/), [**V724**](https://pvs-studio.com/en/docs/warnings/v724/), [**V732**](https://pvs-studio.com/en/docs/warnings/v732/), [**V739**](https://pvs-studio.com/en/docs/warnings/v739/), [**V784**](https://pvs-studio.com/en/docs/warnings/v784/), [**V793**](https://pvs-studio.com/en/docs/warnings/v793/), [**V1019**](https://pvs-studio.com/en/docs/warnings/v1019/),  [**V1029**](https://pvs-studio.com/en/docs/warnings/v1029/),[**V1046**](https://pvs-studio.com/en/docs/warnings/v1046/) |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity)\* | 2017.07 | **NEGATIVE\_RETURNS**  **REVERSE\_NEGATIVE**  **MISRA\_CAST** | Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior  Can find instances where a negativity check occurs after the negative value has been used for something else  Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | ERR34-C | Detect errors when converting as a string to a number |

| **Noncompliant Code** |
| --- |
| This noncompliant code example converts the string token stored in the buff to a signed integer value using the atoi() function: |
| #include <stdlib.h>    void func(const **char** \*buff) {  **int** si;      if (buff) {      si = **atoi**(buff);    } else {      /\* Handle error \*/    }  } |

| **Compliant Code** |
| --- |
| The strtol(), strtoll(), strtoimax()), strtoul(), strtoull(), strtoumax(), strtof(), strtod(), and strtold() functions convert a null-terminated byte string to long int, long long int, intmax\_t, unsigned long int, unsigned long long int, uintmax\_t, float, double, and long double representation, respectively.  This compliant solution uses strtol() to convert a string token to an integer and ensures that the value is in the range of int: |
| #include <errno.h>  #include <limits.h>  #include <stdlib.h>  #include <stdio.h>    void func(const **char** \*buff) {  **char** \*end;  **int** si;    **errno** = 0;      const **long** sl = **strtol**(buff, &end, 10);      if (end == buff) {      (void) **fprintf**(stderr, "%s: not a decimal number\n", buff);    } else if ('\0' != \*end) {      (void) **fprintf**(stderr, "%s: extra characters at end of input: %s\n", buff, end);    } else if ((LONG\_MIN == sl || LONG\_MAX == sl) && ERANGE == **errno**) {      (void) **fprintf**(stderr, "%s out of range of type long\n", buff);    } else if (sl > INT\_MAX) {      (void) **fprintf**(stderr, "%ld greater than INT\_MAX\n", sl);    } else if (sl < INT\_MIN) {      (void) **fprintf**(stderr, "%ld less than INT\_MIN\n", sl);    } else {      si = (**int**)sl;        /\* Process si \*/    }  } |

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

| **Principles(s):** The concept behind this coding standard, ERR34-C - Detect problems while converting a string to a numeric, is "Error Handling and Reporting." This rule underlines the necessity of handling potential mistakes that might happen when converting a string representation to a numeric number and of reporting these issues in a concise and unambiguous way.  How it transfers to this standard is explained as follows:  Error detection is the main focus of the coding standard when it comes to identifying mistakes that may happen when converting a string to a numeric number. The complying code example employs various checks to achieve error detection:  By looking at the end pointer and making sure it points to the string's end with no further characters, one can determine whether the conversion was successful.  By comparing the strtol() result with LONG\_MIN and LONG\_MAX and looking for ERANGE in the errno variable, you may check for possible overflow or underflow scenarios and find issues relating to the range of the resultant number.  determining whether the outcome falls within the int range (INT\_MIN to INT\_MAX).  Reporting errors: An crucial component of error handling is clear and informative error reporting, which is provided by compliant code. When a string cannot be converted into a decimal number, has extra characters, or is outside of the target type's acceptable range, fprintf() is used to print error messages to the standard error stream. These error messages aid programmers in problem diagnosis and resolution.  Error Handling: If an error situation is found, the complying code handles it by reporting it and stopping the conversion. It stays away from the potentially error-prone atoi() function because it does not offer sufficient error reporting options.  The ERR34-C coding standard adheres to the "Error Handling and Reporting" philosophy by highlighting the significance of identifying problems during string-to-numeric conversions, explicitly disclosing these errors, and handling these errors appropriately. The complying code sample uses strtol() and offers thorough error checking and reporting capabilities to illustrate these concepts. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| medium | unlikely | medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | cert-err34-c | Checked by clang-tidy |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Can detect violations of this recommendation by flagging invocations of the following functions:   * + atoi()   + scanf(), fscanf(), sscanf()   Others? |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.2 | **C5030**  **C++5016** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS00-J | Prevent SQL Injection |

| **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](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sanitize) 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** |
| --- |
| This compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long user name. |
| public void doPrivilegedAction(    String username, char[] password  ) throws SQLException {    Connection connection = getConnection();    if (connection == null) {      // Handle error    }    try {      String pwd = hashPassword(password);        // Validate username length      if (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString);      stmt.setString(1, username);      stmt.setString(2, pwd);      ResultSet rs = stmt.executeQuery();      if (!rs.next()) {        throw new SecurityException("User name or password incorrect");      }        // Authenticated; proceed    } finally {      try {        connection.close();      } catch (SQLException x) {        // Forward to handler      }    }  } |

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

| **Principles(s):** This coding standard's guiding principle is called "Prevent SQL Injection." This idea relates to safe coding techniques meant to guard against SQL injection attacks, a prevalent security flaw in database-driven systems.  A type of attack known as SQL Injection allows an attacker to insert erroneous SQL queries into a database by manipulating the application's input. The attacker might be able to enter the system without authorization, retrieve, edit, or remove data, or even compromise the entire database. The coding standard advises the following steps to prevent SQL injection:  Use Parametric Queries: The standard advises against directly encoding user input into SQL statements and instead uses parametric queries where placeholders like "?" are used for dynamic input values in conforming code. As a result, user input is handled as data rather than a component of the SQL query structure.  Before using them in SQL queries, the standard recommends checking user inputs such the length of the username. A check for the username length is included in the complying code to prohibit the usage of excessively long usernames, which might be used to launch attacks.  Sanitize Input: In contrast to the noncompliant code, the compliant approach makes sure that user input, such as passwords, is cleaned up before being used in SQL queries. Instead, before being used in the query, the password is hashed.  Use Prepared Statements: It is advised to use prepared statements for SQL queries (as seen in the complying code with PreparedStatement). It is challenging for attackers to introduce malicious SQL code since they automatically handle argument binding and escaping.  Database connection handling should be done correctly, and the connections should be closed in a finally block to guarantee that resources are released even in the event of exceptions.This coding standard's guiding principle is called "Prevent SQL Injection." This idea relates to safe coding techniques meant to guard against SQL injection attacks, a prevalent security flaw in database-driven systems.  A type of attack known as SQL Injection allows an attacker to insert erroneous SQL queries into a database by manipulating the application's input. The attacker might be able to enter the system without authorization, retrieve, edit, or remove data, or even compromise the entire database. The coding standard advises the following steps to prevent SQL injection:  Use Parametric Queries: The standard advises against directly encoding user input into SQL statements and instead uses parametric queries where placeholders like "?" are used for dynamic input values in conforming code. As a result, user input is handled as data rather than a component of the SQL query structure.  Before using them in SQL queries, the standard recommends checking user inputs such the length of the username. A check for the username length is included in the complying code to prohibit the usage of excessively long usernames, which might be used to launch attacks.  Sanitize Input: In contrast to the noncompliant code, the compliant approach makes sure that user input, such as passwords, is cleaned up before being used in SQL queries. Instead, before being used in the query, the password is hashed.  Use Prepared Statements: It is advised to use prepared statements for SQL queries (as seen in the complying code with PreparedStatement). It is challenging for attackers to introduce malicious SQL code since they automatically handle argument binding and escaping.  Database connection handling should be done correctly, and the connections should be closed in a finally block to guarantee that resources are released even in the event of exceptions. |
| --- |

**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) |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/java/Coverity) | 7.5 | **SQLI FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_** **FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| [Parasoft Jtest](https://wiki.sei.cmu.edu/confluence/display/java/Parasoft) | 2023.1 | **CERT.IDS00.TDSQL** | Protect against SQL injection |
| [Findbugs](https://wiki.sei.cmu.edu/confluence/display/java/Findbugs) | 1.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | CON52-CPP | Prevent data race when accessing bit-field from multiple threads |

| **Noncompliant Code** |
| --- |
| Adjacent bit-fields may be stored in a single memory location. Consequently, modifying adjacent bit-fields in different threads is [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior), as shown in this noncompliant code example. |
| struct MultiThreadedFlags {    unsigned **int** flag1 : 2;    unsigned **int** flag2 : 2;  };    MultiThreadedFlags flags;    void thread1() {    flags.flag1 = 1;  }    void thread2() {    flags.flag2 = 2;  } |

| **Compliant Code** |
| --- |
| This compliant solution protects all accesses of the flags with a mutex, thereby preventing any data races. |
| #include <mutex>    struct MultiThreadedFlags {    unsigned **int** flag1 : 2;    unsigned **int** flag2 : 2;  };    struct MtfMutex {    MultiThreadedFlags s;    std::mutex mutex;  };    MtfMutex flags;    void thread1() {    std::lock\_guard<std::mutex> lk(flags.mutex);    flags.s.flag1 = 1;  }    void thread2() {    std::lock\_guard<std::mutex> lk(flags.mutex);    flags.s.flag2 = 2;  } |

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

| **Principles(s):** This coding standard's and the conforming code's guiding philosophy is "Memory Protection." In computer programming and software development, memory protection is a fundamental idea that tries to assure the safe and proper administration of memory resources, particularly in multi-threaded or concurrent programming environments. The intention is to avoid data races and other memory-related problems that could result in unpredictable behavior and unstable programs.  The goal of this particular coding standard (CON52-CPP) is to prevent data races when several threads access the same bit field. When many threads access and modify shared data simultaneously without sufficient synchronization, data races happen. As a result, various threads may interfere with each other's actions on the same data, which can lead to unpredictable and incorrect behavior.  The complying code uses a mutex, which stands for mutual exclusion, to guard access to the shared bit-fields within the MultiThreadedFlags structure, illustrating how the "Memory Protection" principle is used. Here is how it corresponds to the norm:  Use of Mutex: The complying code adds a member of the MtfMutex structure called mutex to the std::mutex class. The MultiThreadedFlags structure's bit-fields flag1 and flag2 are protected by this mutex.  Mutex Locking: Before changing the bit-fields, a std::lock\_guard is used to acquire and retain the mutex lock (flags.mutex) within each thread function (thread1 and thread2). This prevents data races by ensuring that only one thread can access and modify the bit-fields at once.  The complying code enforces exclusive access to the shared data (flags.s.flag1 and flags.s.flag2) by utilizing the mutex. Consequently, while one thread is updating a bit-field, all additional threads attempting to access or edit the same bit-field will be blocked until the initial thread releases the lock.  Overall, the relevance of memory protection in multi-threaded programming is demonstrated by this coding standard and the complying code. In order to ensure safe access to shared memory resources and prevent data races, it places a strong emphasis on the usage of synchronization methods like mutexes. This enhances program correctness and stability in multi-threaded situations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Coverity) | 6.5 | **RACE\_CONDITION** | Fully implemented |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **read\_write\_data\_race write\_write\_data\_race** | Supported |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-CON52** |  |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: CON52-CPP](https://www.mathworks.com/help/bugfinder/ref/certccon52cpp.html) | Checks for data races (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCLO3-C | Use a static assertion to test the value of a constant expression |

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

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

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

| **Principles(s):** Coding Standard 6, often known as "DCLO3-C," is based on the idea that you should "Use a static assertion to test the value of a constant expression." This coding standard urges programmers to make use of static assertions to check the accuracy of constant expressions at compile time, especially when dealing with variables that are essential for the proper operation of the code.  By using a preprocessor conditional statement (#if) to determine whether a constant expression, in this case the size of a structure (struct timer), matches an anticipated value calculated using other constant expressions (the sizes of its individual members), the compliant code in the example illustrates the principle.  The principle maps to this standard as follows:  The usage of static assertions is encouraged by the idea. Static assertions are commonly implemented in C and C++ programs using preprocessor directives (#if or #error) or the static\_assert keyword.  The standard is primarily concerned with checking constant expressions. Because it depends on the sizes of types and members, which are decided at compile time, the expression being tested is constant in the complying code.  Verification at Compile-Time: The main concept is to carry out the verification at Compile-Time rather than at Runtime. In order to avoid depending on runtime tests, which might be less effective and trustworthy, this is essential for ensuring that vital properties of the code, like the structure's size in this example, are validated during development.  Safety and Correctness: By identifying possible problems with constant expressions early in the development process, the concept and standard attempt to improve code safety and correctness. In this instance, the code makes sure there isn't any padding in the struct timer, which might be crucial to the behavior of the code.  Overall, the principle and standard stress the need of using static assertions to check constant expressions at compile-time, as this enhances code quality, dependability, and maintainability by catching flaws before they arise at runtime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL03** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | Honor exception specifications |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the second function claims to throw only Exception1, but it may also throw Exception2. |
| #include <exception>    class Exception1 : public std::exception {};  class Exception2 : public std::exception {};    void foo() {    throw Exception2{}; // Okay because foo() promises nothing about exceptions  }    void bar() throw (Exception1) {    foo();    // Bad because foo() can throw Exception2  } |

| **Compliant Code** |
| --- |
| This compliant solution catches the exceptions thrown by foo(). |
| #include <exception>    class Exception1 : public std::exception {};  class Exception2 : public std::exception {};    void foo() {    throw Exception2{}; // Okay because foo() promises nothing about exceptions  }    void bar() throw (Exception1) {    try {      foo();    } catch (Exception2 e) {      // Handle error without rethrowing it    }  } |

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

| **Principles(s):** Coding Standard 6, often known as "DCLO3-C," is based on the idea that you should "Use a static assertion to test the value of a constant expression." This coding standard urges programmers to make use of static assertions to check the accuracy of constant expressions at compile time, especially when dealing with variables that are essential for the proper operation of the code.  By using a preprocessor conditional statement (#if) to determine whether a constant expression, in this case the size of a structure (struct timer), matches an anticipated value calculated using other constant expressions (the sizes of its individual members), the compliant code in the example illustrates the principle.  The principle maps to this standard as follows:  The usage of static assertions is encouraged by the idea. Static assertions are commonly implemented in C and C++ programs using preprocessor directives (#if or #error) or the static\_assert keyword.  The standard is primarily concerned with checking constant expressions. Because it depends on the sizes of types and members, which are decided at compile time, the expression being tested is constant in the complying code.  Verification at Compile-Time: The main concept is to carry out the verification at Compile-Time rather than at Runtime. In order to avoid depending on runtime tests, which might be less effective and trustworthy, this is essential for ensuring that vital properties of the code, like the structure's size in this example, are validated during development.  Safety and Correctness: By identifying possible problems with constant expressions early in the development process, the concept and standard attempt to improve code safety and correctness. In this instance, the code makes sure there isn't any padding in the struct timer, which might be crucial to the behavior of the code.  Overall, the principle and standard stress the need of using static assertions to check constant expressions at compile-time, as this enhances code quality, dependability, and maintainability by catching flaws before they arise at runtime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **C++4035, C++4036, C++4632** |  |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **unhandled-throw-noexcept** | Partially checked |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **56 D** | Partially implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | **LANG.STRUCT.EXCP.THROW** | Use of throw |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Internationalization and Localization** | MSC50-CPP | [Do not use std:rand() for generating pseudorandom numbers](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC50-CPP.+Do+not+use+std%3A%3Arand%28%29+for+generating+pseudorandom+numbers) |

| **Noncompliant Code** |
| --- |
| The following noncompliant code generates an ID with a numeric part produced by calling the rand() function. The IDs produced are predictable and have limited randomness. Further, depending on the value of RAND\_MAX, the resulting value can have modulo bias. |
| #include <cstdlib>  #include <string>    void f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    id += std::to\_string(std::**rand**() % 10000);    // ...  } |

| **Compliant Code** |
| --- |
| The C++ standard library provides mechanisms for fine-grained control over pseudorandom number generation. It breaks random number generation into two parts: one is the algorithm responsible for providing random values (the engine), and the other is responsible for distribution of the random values via a density function (the distribution). The distribution object is not strictly required, but it works to ensure that values are properly distributed within a given range instead of improperly distributed due to bias issues. This compliant solution uses the [Mersenne Twister](http://dl.acm.org/citation.cfm?doid=272991.272995) algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the noncompliant code example. |
| #include <random>  #include <string>    void f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    std::uniform\_int\_distribution<**int**> distribution(0, 10000);    std::random\_device rd;    std::mt19937 engine(rd());    id += std::to\_string(distribution(engine));    // ...  } |

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

| **Principles(s):** The coding standard MSC50-CPP, which specifies, "Do not use std::rand() for generating pseudorandom numbers," is based on the idea that, \*\*"Use a high-quality random number generator and distribution for pseudorandom number generation."\*\* In order to guarantee that the generated numbers are actually random and do not display biases or predictability difficulties, this principle underlines the necessity of adopting current and dependable methods for generating pseudorandom numbers in software.  The following is how the idea is effectively addressed in the compliant code that is provided:  High-quality random number generator, for starters The random number generating engine in the complying code is the Mersenne Twister algorithm ('std::mt19937'). A well-known and excellent pseudorandom number generator, the Mersenne Twister is renowned for its lengthy lifespan and strong statistical characteristics. By making this decision, high-quality, unpredictable random numbers are generated.  2. Distribution to Remove Bias The complying code employs a'std::uniform\_int\_distribution' to produce random integers in a given range (in this case, [0-10000]) to overcome the problem of modulo bias that can occur when using the '%' operator with'std::rand()'. By employing this distribution, any bias that might arise while using modulo arithmetic is eliminated and the generated numbers are guaranteed to be evenly distributed throughout the defined range.  3. Using a Random Device to Seed: Additionally, a'std::random\_device' is used in the code to seed the random number generator engine. It is more challenging for an attacker to predict or duplicate the sequence of random numbers generated when a random device is used for seeding since it adds another layer of randomness to the operation.  Overall, the compliant code follows the principle by employing a high-quality random number generator (Mersenne Twister) and a distribution to remove bias, ensuring that the pseudorandom numbers it generates are of good quality and appropriate for a variety of applications, including security-sensitive ones. This method stands in stark contrast to non-compliant code that generates random integers using the antiquated and potentially biased'std::rand()' function. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **bad-function (AUTOSAR.26.5.1A)** | Fully checked |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: MSC50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmsc50cpp.html) | Checks for use of vulnerable pseudo-random number generator (rule partially covered) |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MSC50-a** | Do not use the rand() function for generating pseudorandom numbers |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **44 S** | Enhanced Enforcement |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Testing and Quality Assurance** | EXP54-CPP | [Do not access an object outside of its lifetime](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP54-CPP.+Do+not+access+an+object+outside+of+its+lifetime) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |
| struct S {    void mem\_fn();  };    void f() {    S \*s;    s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, storage is obtained for the pointer prior to calling S::mem\_fn(). |
| struct S {    void mem\_fn();  };    void f() {    S \*s = new S;    s->mem\_fn();    delete s;  } |

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

| **Principles(s):** The Coding Standard EXP54-CPP's prohibition against accessing objects outside of their lifetimes emphasizes the significance of properly managing object lifetimes in C++ to prevent unexpected behavior, and the compliant code example shows how to do this in the right way by making sure that an object is within its valid lifetime before accessing it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: EXP54-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp54cpp.html) | Checks for:   * Non-initialized variable or pointer * Use of previously freed pointer * Pointer or reference to stack variable leaving scope * Accessing object with temporary lifetime   Rule partially covered. |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **42 D, 53 D, 77 D, 1 J, 71 S, 565 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-EXP54-a** **CERT\_CPP-EXP54-b** **CERT\_CPP-EXP54-c** | Do not use resources that have been freed The address of an object with automatic storage shall not be returned from a function The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | **IO.UAC ALLOC.UAF** | Use after close Use after free |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Resource Management** | STR51-CPP. | [Do not attempt to create a std::string from a null pointer](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior) when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    void f() {    std::string tmp(std::**getenv**("TMP"));    if (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    void f() {    const **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");    if (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** This coding standard addresses a resource management philosophy that is frequently linked to robustness and defensive programming ideas. The idea can be succinctly stated as follows:  Don't try to build a std::string from a null pointer, according to the STR51-CPP rule.  It is not recommended to create a std::string object directly from a pointer without first checking for a null pointer, according to this coding standard, specified by STR51-CPP. When you generate a std::string in C++ from a null pointer (such as the outcome of a failed call to std::getenv()), it may behave unexpectedly, cause crashes, or cause other problems with your program.  The std::string object tmp is created in the noncompliant code example directly from the output of std::getenv("TMP"). If the environment variable is absent or if another problem occurs, tmp may store a null pointer as a result. Problems can arise when a std::string is accessed or altered after being constructed from a null pointer.  The standard advises that you first retrieve the output of std::getenv() and assign it to the const char\* variable tmpPtrVal in the example of compliant code. The std::string object tmp is then created using a conditional check to make sure that you can only do so if tmpPtrVal is not null. The method adheres to the standard by preventing the creation of a std::string from a null reference. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR51-a** | Avoid null pointer dereferencing |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C++: STR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr51cpp.html) | Checks for string operations on null pointer (rule partially covered). |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **LANG.MEM.NPD** | Null Pointer Dereference |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **assert\_failure** |  |

### 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.

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Likely | High | Medium | 3 |
| STD-003-CPP | High | Rare | Low | High | 1 |
| STD-004-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-005-CPP | Low | Likely | Low | Low | 1 |
| SYS-001 | High | Rare | Medium | High | 2 |
| SYS-002 | Medium | Moderate | High | Medium | 3 |
| SYS-003 | Low | Likely | Low | Low | 1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | When data is encrypted and kept on non-volatile storage devices like disk drives or databases, this is referred to as encryption in rest. For the purpose of preventing unauthorized access in the event of data breaches or physical theft, this policy is applicable to sensitive data that is at rest.  When stored on servers, databases, or storage devices, all sensitive data must be protected using powerful encryption techniques and properly controlled encryption keys. As a result, even if an attacker has physical access to the storage medium, they will not be able to read the data without the correct decryption keys. |
| Encryption at flight | Securing data while it travels across a network or communication channel is the goal of encryption at flight. To avoid eavesdropping and interception, this policy is applicable to data sent through networks.  Secure transport protocols (like HTTPS and TLS) must be used to encrypt all data sent over open networks or untrusted channels. This guarantees that information stays private and essential while in transit, guarding it against illegal access or alteration. |
| Encryption in use | While data is being processed or used by programs, it is protected by encryption. Applications, APIs, and services that process data are covered by this policy.  Applications should use encryption to safeguard sensitive data while it is being processed. This includes secure key management, encryption of important configuration settings, and encryption of data stored in memory. This stops hackers from stealing private data from active applications. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying the identity of persons or systems trying to access resources is the process of authentication. This policy makes sure that only individuals with permission can access the systems and data.  Before acquiring access to sensitive information, all users and systems must go through a rigorous authentication process, such as multi-factor authentication (MFA). This guards against identity theft and prevents unwanted access. |
| Authorization | After successful authentication, authorization specifies what actions or resources a user or system is permitted to access. It enacts permissions and access controls.  Implementing authorization policies will allow you to limit access to resources according to user roles and permissions. This reduces the possibility of data breaches and illegal acts by ensuring that users can only access information related to their roles. |
| Accounting | Accounting include the monitoring and recording of all actions, including events involving authentication and authorisation. For reasons of compliance and monitoring, it offers an audit trail.  Mechanisms for thorough logging and auditing must be in place to keep track of user actions, attempted access, and system events. This helps in locating and looking into security incidents, preserving compliance, and guaranteeing accountability. |

**\***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 |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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