**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 | Establishing stringent input validation practices across applications, encompassing various input sources such as network data, keyboard inputs, file interactions, and database queries, is essential. The validation process should be thorough, only allowing permitted recognized and trusted input types while diligently filtering out data from untrusted sources.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Heed Compiler Warnings | To ensure the utmost code quality and security, compile code with the highest warning level supported by the compiler. Address warnings by modifying the code and following C MSC00-A and C++MSC00-A guidelines. Use static and dynamic analysis tools to detect and fix flaws. This proactive approach helps catch defects early, preventing potential chain reactions of failures. Treating compiler warnings as errors is wise since they often reveal underlying code problems despite varying industry options.  Source: http://web.archive.org/web/20160404152049/https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices |
| 1. Architect and Design for Security Policies | Architecting and designing security policies formalize rules for an organization’s security structure, specifying roles for asset protection. This architecture includes people, processes, and tools. Good architecture enhances code uniqueness and security through secure coding, subsystems, and trusted libraries. When creating software architecture, ensure effective security enforcement, such as dividing the system into subsystems with appropriate privilege levels as needed.  Source: http://web.archive.org/web/20160404152049/https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices |
| 1. Keep It Simple | Adhering to the “Keep It Simple” (KISS) principle is crucial. This guideline stresses the significance of maintaining a straightforward, small design since complex designs are more prone to errors. KISS, a well-recognized principle in programming, promotes simplicity while ensuring system efficiency. This approach simplifies coding, improving transparency, efficiency, and security while keeping user-friendly code and avoiding unnecessary complexity in security controls.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Default Deny | Base access on permissions, not exclusion, by adopting the “Default Deny” concept. Assume code security, challenge vulnerabilities until proven, and refrain from user issue acknowledgment or accepting responsibility for imperfections. Implement a default deny policy to keep unauthenticated users out and apply it to new user accounts and features.    Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Adhere to the Principle of Least Privilege | Embrace the Principle of Least Privilege (PoLP), which entails restricting users or entities to the minimum necessary privileges, reducing elevated permissions and security risks. To effectively implement PoLP, validate permissions for routine requests, establish permission validation tests before release, and conduct periodic permission reviews.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Sanitize Data Sent to Other Systems | Ensure data from untrusted sources transferred to an output subsystem is sanitized to prevent injection attacks. Output sanitization is equally vital as robust input validation. Consistent data sanitization is essential to prevent misuse of unused features in output data. Apply contextual escaping the code responsible for output generation. Prioritize data sanitization before sending inputs to other systems using practices like whitelists, blacklists, and input escaping. Sanitize data passed to complex subsystems, like command shells, databases, and COTS components, as the calling process ensures data integrity.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Practice Defense in Depth | Defense-in-depth (DiD) is a multi-layered cybersecurity strategy that deploys multiple defensive measures through the IT infrastructure from diverse attack vectors. It sets up multiple tiers of network and system defenses to efficiently counter advanced attacks. DiD includes administrative, physical, and technical controls tailored for safeguarding the network. Employing multiple defense strategies helps mitigate risks, ensuring that if one layer is insufficient, another can prevent vulnerabilities from being exploited or limit their consequences. For example, combining secure programming techniques with secure runtime environments reduces the likelihood of exploiting code vulnerabilities in operational environments.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Use Effective Quality Assurance Techniques | To enhance the robustness of the software systems, it is essential to employ potent quality assurance methods such as penetration testing, source code audits, fuzz testing, and application scanning. These techniques are crucial in pinpointing and subsequently mitigating vulnerabilities within the applications. Integrating application scanning into a development process is recommended, particularly before implementing significant changes or revisions slated for production. By doing so, one can proactively identify and rectify any potential security issues, ensuring the resilience and security of the software throughout its lifecycle.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |
| 1. Adopt a Secure Coding Standard | Create or implement a coding standard focused on security for a specific programming language and development platform.  Source: https://safecomputing.umich.edu/protect-the-u/protect-your-unit/secure-coding/best-practices |

### 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** | [STD-001-CPP] | DCL59-CPP. Do not define an unnamed namespace in a header file. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL59-CPP.+Do+not+define+an+unnamed+namespace+in+a+header+file

| **Noncompliant Code** |
| --- |
| The noncompliant code attempts to resolve the link-time errors by defining f()within an unnamed namespace. However, it produces multiple, unique definitions of f()in the resulting executable. If a.h is included from many translation units, it can lead to increased link times, a larger executable file, and reduced performance. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    namespace {  void f() { /\* ... \*/ }  }    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  // ...    // b.cpp  #include "a.h"  // ... |

| **Compliant Code** |
| --- |
| In this compliant solution, f()is not defined with an unnamed namespace and is instead defined as an inline function. Inline functions are required to be defined identically in all the translation units in which they are used, which allows an implementation to generate only a single instance of the function at runtime in the body of function does not get generated for each call site. |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE    inline void f() { /\* ... \*/ }    #endif // A\_HEADER\_FILE    // a.cpp  #include "a.h"  // ...    // b.cpp  #include "a.h"  // ... |

**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**  Adhering to “Keep It Simple” and “Use Effective Quality Assurance Techniques” for DCL59-CPP involves avoiding unnecessary complexity by refraining from defining unnamed namespaces in header files. Utilizing a combination of static analysis tools and manual reviews, enhancing overall code quality and maintainability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Unnamed-namespace-header | Fully checked |
| Clang | 3.9 | Cert-dcl59-cpp | Checked by clang-tidy |
| LDRA tool suite | 9.7.1 | 286 S, 512 S | Fully implemented. |
| Polyspace Bug Finder | R2023a | CERT C++:DCL59-CPP | Checks for unnamed namespaces in header files (rule fully covered). |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | EXP50-CPP. Do not depend on the order of evaluation for side effects. |

Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP50-CPP.+Do+not+depend+on+the+order+of+evaluation+for+side+effects

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, i is evaluated more than once in an unsequenced manner, so the behavior of the expression is undefined. |
| void f(**int** i, const **int** \*b) {  **int** a = i + b[++i];    // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution is independent of the order of evaluation of the operands and can be interpreted in only one way. |
| void f(**int** i, const **int** \*b) {    ++i;  **int** a = i + b[i];    // ...  } |

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

| **Principles(s):**  **1. Validate Input Data**  **4. Keep it Simple**  **9. Use Effective Quality Assurance Techniques**  The CERT C++ Coding Standard rule EXP50-CPP, advises against replying on the order of evaluation for side effects, aligns with secure coding principles “Validate Input Data” and indirectly supports code simplicity, in line with “Keep It Simple.” Additionally it adherence to “Use Effective Quality Assurance Techniques” is essential for ensuring compliance. Rigorous quality assurance practices, including code reviews and potential static analysis, are crucial in identifying and addressing dependencies on the order of evaluation, contributing to elevated code quality and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wunsequenced | Can detect simple violations of this rule where path-sensitive analysis is not required. |
| CodeSonar | 7.4p0 | LANG.STRUCT.SE.DEC | Side Effects in Expression with Decrement  Side Effects in Expression with Increment |
| Coverity | V7.5.0 | EVALUATION\_ORDER | Can detect the specific instance where a statement contains multiple side effects on the same value with an undefined evaluation order because, with different compiler flags or different compilers or platforms, the statement may behave differently. |
| Polyspace Bug Finder | R2023a | CERT C++:EXP50-CPP | Checks for situations where expression value depends on order of evaluation (rule fully covered). |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the unformatted input function std::basic\_istream<T>::read()is used to read an unformatted character array of 32 characters from the given file. However, the read()function does not guarantee that the string will be null terminated, so the subsequent call of the std::string constructor results in undefined behavior if the character array does not contain a full terminator. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  **char** buffer[32];    try {      in.read(buffer, sizeof(buffer));    } catch (std::ios\_base::failure &e) {      // Handle error    }      std::string str(buffer);    // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the std::string object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use std::basic\_istream<T>::readsome()or a formatted input function, depending on need. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  **char** buffer[32];    try {      in.read(buffer, sizeof(buffer));    } catch (std::ios\_base::failure &e) {      // Handle error    }    std::string str(buffer, in.gcount());    // ...  } |

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

| **Principles(s):**  **4. Keep It Simple**  **9. Use Effect Quality Assurance Techniques**  **10 Adopt a Secure Coding Standard**  The STR50-CPP rule, emphasizing sufficient storage for strings to prevent vulnerabilities, aligns with the principles of “Use Effective Quality Assurance Techniques” through rigorous testing and code reviews. It also supports “Keep It Simple” by promoting straightforward string-handling practices and falls under the umbrella of “Adopt a Secure Coding Standard.” This approach ensures secure and reliable string manipulation in software development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stream-input-char-array | Partially check + soundly supported. |
| CodeSonar | 7.4p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator.  Buffer overrun  Type overrun |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string.  Avoid overflow when writing to a buffer.  Prevent buffer overflows from tainted data.  Avoid buffer write overflow from tainted data.  Do not use the ‘char’ buffer to store input from ‘std::cin’ |
| Polyspace Bug Finder | R2023a | CERT C++:STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | IDS00-J. Prevent SQL injection. |

**Source:** <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection>

| **Noncompliant Code** |
| --- |
| The provided code example illustrates JDBC code for user authentication in a system. This noncompliant code passes the password as a char array, and after creating the database connection, it hashes the passwords. The vulnerability arises from incorporating the unsanitized input argument, the username, into the SQL command, enabling the possibility of an SQL injection attack with input like validuser’ OR ‘1’=’1. The password argument remains protected as it undergoes hashing within the hashPassword () function, including input sanitization. |
| 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):**  **1. Validate Input Data**  **4. Architect and Design for Security Policies**  **5. Default Deny**  **7. Sanitize Data Sent to Other Systems**  **9. Use Effective Quality Assurance Techniques**  **10. Adopt a Secure Coding Standard**  The SQL injection prevention standard IDS00-J aligns with multiple secure coding principles. It emphasizes input data validation, supporting “Validate Input Data” and “Architect and Design for Security Policies.” By promoting the denial of access to vulnerable inputs, IDS00-J aligns with “Default Deny.” It also emphasizes data sanitization before transmission, aligning with “Sanitize Data Sent to Other Systems.” Additionally, IDS00-J advocates rigorous testing and code reviews, supporting “Use Effective Quality Assurance Techniques.” As a specific coding standing, IDS00-J exemplifies “Adopt a Secure Coding Standard,” ensuring consistent and reliable practices for SQL injection prevention. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Parasoft Jtest | 2023.1 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SpotBugs | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE  SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | MEM52-CPP. Detect and handle memory allocation errors. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using::operator new[] (std::size\_t)and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions.  Because ::operator new[] std::size\_t)can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    void f(const **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = new **int**[size];    std::**memcpy**(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    void f(const **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = new (std::nothrow) **int**[size];    if (!copy) {      // Handle error      return;    }    std::**memcpy**(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

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

| **Principles(s):**  **9.Use Effective Quality Assurance Techniques**  **10. Adopt a Secure Coding Standard**  MEM52-CPP, focusing on detecting and handling memory allocation errors, aligns with the coding principles of “Use Effective Quality Assurance Techniques” and “Adopt a Secure Coding Standard.” The rule underscores the importance of rigorous quality assurance practices, such as thorough testing and code reviews, to identify and address potential memory allocation errors during development. Additionally, MEM52-CPP is a specific guideline within the broader context of secure coding standards, emphasizing adopting consistent and secure practices for detecting and handling memory allocation issues. Collectively, these principles contribute to creating robust and reliable code, particularly in systems involving memory management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled. |
| LDRA tool suite | 9,7,1 | 45 D | Partially implemented. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new. Do not allocate resources in function argument list because the order of evaluation of a function’s parameters is undefined. |
| Polyspace Bug Finder | R2023a | CERT C++: MEM52\_CPP | Check for unprotected dynamic memory allocation (rule partially covered). |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | DCL03-C. Use a static assertion to test the value of a constant expression. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/c/DCL03-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 ma7 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):**  **9. Use Effect Quality Assurance Techniques**  **10. Adopt a Secure Coding Standard**  DCL03-C “Use a static assertion to test the value of a constant expression” recommends employing static assertions for testing constant expressions, aligning with the principles of “Use Effective Quality Assurance Techniques” and “Adopt a Secure Coding Standard.” This guideline underscores the significance of robust quality assurance practices and adherence to secure coding standards, ensuring reliable validation of constant expressions in the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.4p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented. |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | ERR50-CPP. Do not abruptly terminate the program |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR50-CPP.+Do+not+abruptly+terminate+the+program

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(),may result in a call to std::terminate()because throwing\_func()may throw an exception. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.    throwing\_func();  }    **int** main() {    if (0 != std::**atexit**(f)) {      // Handle error    }    // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f()handles all exceptions thrown by throwing\_func()and does not rethrow. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.    try {      throwing\_func();    } catch (...) {      // Handle error    }  }    **int** main() {    if (0 != std::**atexit**(f)) {      // Handle error    }    // ...  } |

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

| **Principles(s):**  **4. Keep It Simple**  **8. Practice Defense in Depth**  **9. Use Effective Quality Assurance Techniques**  **10. Adopt a Secure Coding Standard**  The coding principle “Practice Defense in Depth aligns directly with ERR50-CPP, which advises against abruptly terminating the program. This principle advocates for implementing multiple layers of defense mechanisms, emphasizing the importance of handling errors to avoid insecure states in the system. Additionally, other principles contribute indirectly to the goal of avoiding abrupt program termination: “Keep It Simple” promotes straightforward code to reduce the likelihood of errors. “Use Effective Quality Assurance Techniques” ensures rigorous testing and code reviews to identify potential issues, and “Adopt a Secure Coding Standard” supports following guidelines for error handling and program termination. Together, these principles collectively contribute to code quality, robustness, and the prevention of abrupt program terminations as advised by ERR50-CPP. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stdlib-use | Partially checked |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced Enforcement |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR0-a  CERT\_CPP-ERR0-b  CERT\_CPP-ERR0-c  CERT\_CPP-ERR0-d  CERT\_CPP-ERR0-e  CERT\_CPP-ERR0-f  CERT\_CPP-ERR0-g  CERT\_CPP-ERR0-h  CERT\_CPP-ERR0-i  CERT\_CPP-ERR0-j  CERT\_CPP-ERR0-k  CERT\_CPP-ERR0-l  CERT\_CPP-ERR0-m  CERT\_CPP-ERR0-n | The execution of a function registered with ‘std::atexit()’ or ‘std::at\_quick\_exit()’ should not exit via an exception.  Do not throw form within destructor.  There should be at least one exception handler to catch all otherwise unhandled exceptions.  An empty throw (throw;) shall only be used in the compound-statement of a catch handler.  Exceptions shall be raised only after start-up and before termination of the program.  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.  Where a function’s declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).  Function called in global or namespace scope shall not throw unhandled exception.  Always catch exceptions.  Properly define exit handlers.  The ‘about()’ function from the ‘stdlib.h’ or ‘cstdlib’ library shall not be used.  Avoid throwing exceptions from functions that are declared not to throw.  The ‘quick\_exit()’ and ‘\_Exit()’ functions from the ‘stdlib.h’ or ‘cstdlib’ library shall not be used. |
| Polyspace Bug Finder | R2023a | CERT C++:ERR50-CPP | Checks for implicit call to terminate () function (rule partially covered). |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Expressions** | [STD-008-CPP] | EXP51-CPP. Do not delete an array through a pointer of the incorrect type. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP51-CPP.+Do+not+delete+an+array+through+a+pointer+of+the+incorrect+type

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of Derived objects is created and the pointer is stored in a Base \*. Despite Base::~Base()being declared virtual, it is still results in undefined behavior. Further, attempting to perform pointer arithmetic on the static type Base \* violates CTR56-CPP. Do not use point arithmetic on polymorphic objects. |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Base \*b = new Derived[10];     // ...     delete [] b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the static type of b is Derived \*, which removes the undefined behavior when indexing into the array as well as when deleting the pointer. |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Derived \*b = new Derived[10];     // ...     delete [] b;  } |

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

| **Principles(s):** E  **8. Practice Defense in Depth**  **9. Use Effective Quality Assurance Techniques**  **10. Adopt a Secure Coding Standard**  EXP51-CPP, cautioning against deleting arrays through incorrect pointer types, aligns primarily with the “Practice Defense in Depth” principle. This principle advocates for layered defense mechanisms to bolster software security. Additionally, EXP51-CPP resonates with “Use Effective Quality Assurance Techniques” and “Adopt a Secure Coding Standard,” promoting rigorous testing, code reviews, and adherence to secure coding standards. Together, these principles form a comprehensive approach to prevent errors, specifically those related to incorrect array deletions through pointers of the wrong type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -analyzer-  checker=cplusplus | Checked with clang –ccl  Or (preferably) scan-build |
| CodeSonar | 7.4p0 | ALLOC.TM | Type Mismatch |
| Parsoft C/C++test | 2023.1 | CERT\_CPP-EXP51-a | Do not treat arrays polymorphically |
| Polyspace Bug Finder | R2023a | CERT C++:EXP51-CPP | Checks for delete operator used to destroy downcast object of different type. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Declarations and Initializations (DCL)** | [STD-009-CPP] | DCL58-CPP. Do not modify the standard namespaces. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL58-CPP.+Do+not+modify+the+standard+namespaces

| **Noncompliant Code** |
| --- |
| In the noncompliant code example, the declaration of x is added to the namespace std, resulting in undefined behavior. |
| namespace std {  **int** x;  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes the intention of the programmer was to place the declaration of x into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name. |
| namespace nonstd {  **int** x;  } |

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

| **Principles(s):**  **4. Keep It Simple**  **10. Adopt a Secure Coding Standard**  DCL58-CPP cautions against modifying standard namespaces, aligning closely with the “Keep It Simple” and “Adopt a Secure Coding Standard.” This guideline encourages developers to maintain code simplicity by avoiding unnecessary changes to standard namespaces, minimizing potential complexities and conflicts. Additionally, adhering to DCL48-CPP contributes to adopting secure coding standards, ensuring code portability, compatibility, and adherence to best practices. Following this guideline, developers contribute to a clear, straightforward codebase that prioritizes simplicity and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.STRUCT.DECL.SNM | Modification of Standard Namespaces |
| Helix QAC | 2023.3 | C++3180, C++3181, C++3182 |  |
| Parasoft  C/C++test | 2023.1 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces ‘std’ and ‘posix’ |
| Polyspace Bug Finder | R2023a | CERT C++:DCL58-CPP | Checks for modification of standard namespaces (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Object Oriented Programming (OOP)** | [STD-010-CPP] | OOP56-CPP. Honor replacement handler requirements. |

**Source:** https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP56-CPP.+Honor+replacement+handler+requirements

| **Noncompliant Code** |
| --- |
| In this noncompliant cod example, a replacement new\_handler is written to attempt to release salvageable resources when the dynamic memory manager runs out of memory. However, this example does not take into account the situation in which all salvageable resources have been recovered and there is still insufficient memory to satisfy the allocation request. Instead of terminating the replacement handler with an exception of type std::bad\_alloc or terminating the execution of the program without returning to the caller, the replacement handler returns as normal. Under low memory conditions, an infinite loop will occur with the default implementation of ::operator new(). Because such conditions are rare in practice, it is likely for this bug to go undiscovered under typical testing scenarios. |
| #include <new>    void custom\_new\_handler() {    // Returns number of bytes freed.    extern std::**size\_t** reclaim\_resources();    reclaim\_resources();  }    **int** main() {    std::set\_new\_handler(custom\_new\_handler);      // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, custom\_new\_handler()uses the return value from reclaim\_resources(). If it returns 0, then there will be insufficient memory for operator new to succeed. Hence, an exception of type std::bac\_alloc is thrown, meeting the requirements for the replacement handler. |
| #include <new>    void custom\_new\_handler() noexcept(false) {    // Returns number of bytes freed.    extern std::**size\_t** reclaim\_resources();    if (0 == reclaim\_resources()) {      throw std::bad\_alloc();    }  }    **int** main() {    std::set\_new\_handler(custom\_new\_handler);      // ...  } |

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

| **Principles(s):**  **8. Practice Defense in Depth**  **9. Use Effective Quality Assurance Techniques**  **10. Adopt a Secure Coding Standard**  OOP56-CPP, emphasizing the importance of honoring replacement handler requirements in object-oriented programming, aligns with secure coding principles. Specifically, “Practice Defense in Depth” supports replacement handlers as a layer of defense, and “Use Effective Quality Techniques” highlights rigorous testing for correct implementation. Furthermore, OOP56-CPP embodies “Adopt a Secure Coding Standard,” emphasizing consistent adherence to replacement handler specifications within secure coding standards. Collectively, these principles ensure a secure, reliable, and thoroughly tested approach to handling errors and exceptions in code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 20233 | DF4776, DF4777, DF4778, DF$779 |  |
| Parasoft  C/C++ test | 2023.1 | CERT\_CPP-OOP56-a  CERT\_CPP-OOP56-b  CERT\_CPP-OOP56-c | Properly define terminate handlers.  Properly define unexpected handlers.  Properly define new handlers. |
| Polyspace Bug Finder | R2023b | CERT C++ OOP56-CPP | Checks for replacement handler function that does not meet requirements (rule fully covered). |
| SonarQube | 10.3 |  | This open-source platform supports continuous inspection of code quality and can be configured to check for compliance with replacement handler requirements and other coding standards in C++. |

### 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 | Medium | Unlikely | Medium | P4 (Low) | L3 |
| STD-002-CPP | Medium | Probable | Medium | P8 (High) | L2 |
| STD-003-CPP | High | Likely | Medium | P18 (High) | L1 |
| STD-004-CPP | High | Likely | Medium | P18 (High) | L1 |
| STD-005-CPP | High | Likely | Medium | P18 (High) | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 (Low) | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 (Low) | L3 |
| STD-008-CPP | Low | Unlikely | Medium | P2 (Low) | L3 |
| STD-009-CPP | High | Unlikely | Medium | P6 (Medium) | L2 |
| STD-010-CPP | Low | Probable | High | P2 (Low) | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Data encryption at rest provides a protective layer for stored information, encompassing various storage mediums like hard drives, phones, computers, and cloud assets. Safeguarding this data involves employing encryption tools, implementing disk encryption, and ensuring security measures for mobile devices and computers. |
| Encryption at flight | Encryption in transit pertains to safeguarding data while it is in motion. It involves protecting data as it moves between devices within a network or traverses outside of a network. Security measures for this type of encryption include email encryption, Data Loss Prevention (DLP) solutions, and robust network security features such as firewalls and authentication. It is also imperative to consider and ensure the security of the transmission path that the data follows. |
| Encryption in use | Encryption safeguards data generated, edited or otherwise classified as actively in use. Protecting this data involves implementing data control and protection measures before and during its utilization. Managing access rights and identities is crucial in minimizing risks associated with this data. |

source:. <https://www.codingdojo.com/blog/secure-coding-practices>

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication involves verifying an individual’s identity through various methods such as static passwords, one-time passwords, certifications, and biometric credentials. These various forms of identification confirm that an individual is who they claim to be. |
| Authorization | Authorization defines a user’s access rights and privileges, crucial to information and computer security. While authentication verifies an identity, authorization dictates the user’s permissible access, mitigating potential vulnerabilities by restricting interactions with sensitive data that may be unnecessary or defining the permissions granted during access. |
| Accounting | Accounting involves tracking user activity within a system, noting timestamps, accessed resources, and data transfer information. This process is vital for forensic analysis and investigation when necessary. |

Source: <https://www.geeksforgeeks.org/computer-network-aaa-authentication-authorization-and-accounting/>

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 11/11/2023 | Module 3 Milestone | Richard Backscheider |  |
| [Insert text.] | 12/3/2023 | Module 6 Milestone | Richard Backscheider |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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