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

std

# 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 | Data that is not validated and sanitized can lead to improper input, causing software vulnerabilities. Hence, developers can combat this by validating data. |
| 1. Heed Compiler Warnings | Compiler warnings can highlight some errors and issues found within code that may have been missed. Going through and eliminating the mistakes will make the code more secure than if the warning was left alone. |
| 1. Architect and Design for Security Policies | Designing with security policies in mind will make integration seamless, prioritize accessibility, and create a security-conscious design that is also user-friendly and intuitive design. In addition, this approach anticipates and mitigates risk by designing securely. |
| 1. Keep It Simple | The more complicated a design gets, the higher the chance for security vulnerabilities to be present. Keeping the overall design simple will make designing and implementing security changes effortlessly while making maintenance a breeze. |
| 1. Default Deny | Design the program only to provide access to required information and deny everything else unless those parameters are met. |
| 1. Adhere to the Principle of Least Privilege | Only allow users or entities access to specific data necessary to complete the required tasks to eliminate vulnerability risks. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it is vital to ensure it remains secure. This principle mitigates potential vulnerabilities by sanitizing and validating data before it is sent. |
| 1. Practice Defense in Depth | Multiple security layers that overlap will make it much harder for a hacker to gain information. In addition, this approach ensures that if one of the defenses is breached, the data is still safe and guarded by another. |
| 1. Use Effective Quality Assurance Techniques | Utilize different quality assurance techniques to identify vulnerabilities early and correct them before the program goes live. This is a proactive measure that will help eliminate data leaks and attacks in the future. |
| 1. Adopt a Secure Coding Standard | Secure coding standards ensure resilient software is being developed. By establishing guidelines, developers can ensure the code is functional, secure, and fortified against threats. In addition, this will allow for any security changes or upgrades to be implemented much more efficiently and smoothly. |

### 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-CLG | Implement abstract data types using opaque types  Abstract data types are not limited to object-oriented languages like CPP and Java, and they can be used in many languages, especially C. Abstract data types are especially useful when used alongside private (opaque) data types that conceal information. |

| **Noncompliant Code** |
| --- |
| This noncompliant code, shown below, is derived from the CERT-managed string library and exposes the implementation of the "string\_mx" type in the 'string\_m.h' header file. The problem with this visibility is that it increases the risk of violating software principles like information hiding and data encapsulation, which can lead to the development of defective or nonportable code. |
| **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;  unsigned **char** strtype;  **char** \*cstr;  };    **typedef** **struct** string\_mx string\_mx;    /\* Function declarations \*/  **extern** errno\_t strcpy\_m(string\_mx \*s1, **const** string\_mx \*s2);  **extern** errno\_t strcat\_m(string\_mx \*s1, **const** string\_mx \*s2);  /\* ... \*/ |

| **Compliant Code** |
| --- |
| This approach redefines the 'string-mx' type to private, which will conceal the data type's implementation from users of the managed string library. In addition, within the header file, the definition of 'struct string-mx' is present but remains hidden from users interacting with the data due to abstraction. |
| **struct** string\_mx;  **typedef** **struct** string\_mx string\_mx;    /\* Function declarations \*/  **extern** errno\_t strcpy\_m(string\_mx \*s1, **const** string\_mx \*s2);  **extern** errno\_t strcat\_m(string\_mx \*s1, **const** string\_mx \*s2); |
| /\* ... \*/  **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;  unsigned **char** strtype;  **char** \*cstr;  }; |

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

| **Principles(s): Architect and Design for Security Policies (3), and Adopt a Secure Coding Standard(3).**  In the noncompliant example, the struct is not private or hidden. It is fully visible to the user and anyone using the struct. Coding this way allows the data to be directly manipulated because it is not encapsulated or hidden. Therefore, we must follow the architect and design for security policy to help avoid this. Also, adopt a secure coding standard is essential because encapsulation of sensitive data is critical to secure programming so that it stays protected. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL12 | N/A |
| LDRA Tool Suite | 9.7.1 | 104 D | Partially Implemented |
| Polyspace Bug Finder | R2022a | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |
| Parasoft C/C++ test | 2022.1 | CERT\_C-DCL12-a | If a pointer to a structure or union is never dereferenced withing a translation unit, then the implementation of the object should be hidden |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not cast to an out-of-range enumeration value  By not casting values to an out-of-range enumeration type, we can ensure we adhere to the intended design of enum constructs, ensure type safety and withheld, and promote code clarity. In addition, making sure this standard is followed will minimize the risk of undefined behavior, contributing to more maintainable and predictable code. |

| **Noncompliant Code** |
| --- |
| This example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value |
| **enum** EnumType {  First,  Second,  Third  };    **void** f(**int** intVar) {  EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| **enum** EnumType {  First,  Second,  Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** **Validate Input Data**  Verify all input values are in range before casting to the enumeration type to reduce the likelihood of casting to an out-of-range value. Not verifying input can lead to unspecified behavior, causing problems now or later in the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | N/A |
| CodeSonar | 7.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| RuleChecker | 22.10 | Cast-integer-to-enum | Partially checked |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator  Copying data to a string with insufficient space can lead to a buffer overflow, causing unwanted parties to gain access to data and information, which is a huge security risk. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    **void** f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** **Architect and Design for Security Policies(3), and Adopt a Secure Coding Standard(10):**  Make sure that strings are validated. Verify strings have sufficient space before copying data, avoiding unwanted behaviors or issues like buffer overflow. A buffer overflow allows attackers to exploit the data, causing further unwanted problems. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stream-input-char-array | Partially checked + soundly supported |
| CodeSonar | 7.4p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Helix QAC | 2023.3 | C++5216  DF2835, DF2836, DF2839 | N/A |
| LDRA tool suite | 9.7.1 | 478 S, 66 X, 70 X, 71 X | Partially Implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-JAV | Prevent SQL Injection  When input for an SQL query comes from an untrusted source, SQL injection risks can lead to data vulnerabilities. However, this can be prevented with input sanitization and validation, ensuring data is not malicious or incorrect. |

| **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. |
| 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 username. |
| 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):** **Validate Input Data(1), Heed Compiler Warnings(2), Architect and Design for Security Policies(3), Default Deny(5), Sanitize Data Sent to other Systems(7), Practice Defense in Depth(8), Use Effective Quality Assurance Techniques(9), and Adopt a Secure Coding Standard(10):**  Validate all input to eliminate any potential malicious characters.  Employ an analysis tool to detect query warnings, hopefully mitigating the risk of any vulnerabilities leading to injection.  Design the program architecture to defend against SQL injection attacks proactively.  Restrict access to data unless the correct password is provided, and any other specified conditions are met.  Consistently sanitize, validate, and authenticate user input, especially passwords, before passing them to any part of the program.  Use Defense in Depth strategies to help safeguard data, ensuring a robust security measure against any successful SQL injection attempts.  Conduct thorough testing throughout development to check the code's security measures.  Establish a coding standard for programmers to proactively scan for malicious user input and deny any unauthorized access to data as a core principle. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 7.4p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access freed memory  Accessing released memory can result in unpredictable behavior and several runtime issues, violate memory management guidelines, and jeopardize the program's stability. It is crucial for any program to properly handle memory deallocation and refrain from accessing it afterward to maintain a reliable and secure program. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {  S \*s = **new** S;  // ...  **delete** s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** Heed Compiler Warnings(2), Sanitize Data Sent to Other Systems (7), Practice Defense in Depth (8), and Adopt a Secure Coding Standard(10);  Attempting to access freed memory will trigger warnings in the analyzers and compile time checker, serving as a crucial indicator of potential issues.  When a program is coded to free and delete memory after proper utilization, the transmission to another system or function must also be sanitized and restricted.  If memory is accessed after being freed, the data should be encrypted. Encryption serves as a safeguard preventing the extraction of sensitive information if accessed through memory.  Developers must be skilled in secure memory management. Comprehensive training should be done to ensure they know how to utilize and release memory to enhance code security properly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | ALLOC.UAF | Use after free |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| LDRA tool suite | 9.7.1 | 483 S , 484 S | Partially implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG | Use a static assertion to test the value of a constant expression  Using a static assertion to test the value of constant expressions in programming languages allows us to implement a compile-time validation test. Assertions improve code reliability by asserting it matches the expected outcome, minimizing risks associated with improper logic or initializations. |

| **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** |
| --- |
| This portable compliant solution uses static\_assert: |
| #include <assert.h>    **struct** timer {  unsigned **char** MODE;  unsigned **int** DATA;  unsigned **int** COUNT;  };    static\_assert(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**),  "Structure must not have any padding"); |

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

| **Principles(s):** Architect and Design for Security Policies (3):  Take a proactive approach to security by assessing constant values. This practice will help streamline and detect errors early during compile time, maximizing efficiency during development. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 | N/A |
| 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 |
| Compass/ROSE | N/A | N/A | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions thrown before main() begins executing  Handling exceptions thrown before main() is executed allows us to ensure a smooth program initialization. This approach ensures any potential issues are resolved early on and exceptions do not go uncaught. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the constructor for S may throw an exception that is not caught when globalS is constructed during program startup. |
| **struct** S {  S() noexcept(**false**);  };    **static** S globalS; |

| **Compliant Code** |
| --- |
| This compliant solution makes globalS into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for S will be executed the first time the function globalS() is called rather than at program startup. This solution does require the programmer to modify source code so that previous uses of globalS are replaced by a function call to globalS(). |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** Architect and Design for Security Policies (3), and Adopt a secure Coding Standard(10):  Incorporate exception handling into the design to prevent unattended behavior or data leaks.  Instill in developers the practice of incorporating and managing exception-handling techniques to uphold our secure coding standards. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program |
| Polyspace Bug Finder | R2023A | CERT C++: ERR58-CPP | Checks for exceptions raised during program start up (rule fully covered) |
| RuleChecker | 22.10 | Potentially-throwing-static-initialization | Partially Checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Management** | STD-008-CPP | Detect and handle memory allocation errors  Identifying and managing memory allocation errors is crucial for maintaining the program's reliability and stability. Using error detection techniques in our code, the program will be better suited to handle memory allocation issues, mitigating potential disruptions, errors, and vulnerabilities. |

| **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):** Heed Compiler Warnings (2), Architect and Design for Security Policies (3), Adopt a Secure Coding Standard (10):  Using a reliable analysis tool will allow us to detect any memory errors, hopefully preventing any.  Incorporating features in the program's design to identify and address memory errors will help prevent any issues later.  Ensuring that developers possess sufficient skill in error handling will help them understand how to implement standards effectively. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**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 | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-009-CLG | Use correct integer precisions  Accurate integer precision is crucial for ensuring numerical representation in programming is accurate and no unintended data loss or overflow happens when using these values. This practice will ensure optimal memory allocation and make sure all mathematical operations are precise. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example illustrates a function that produces 2 raised to the power of the function argument. To prevent undefined behavior in compliance with INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand, the function ensures that the argument is less than the number of bits used to store a value of type unsigned int. However, if this code runs on a platform where unsigned int has one or more padding bits, it can still result in values for exp that are too large. For example, on a platform that stores unsigned int in 64 bits, but uses only 48 bits to represent the value, a left shift of 56 bits would result in undefined behavior. |
| #include <limits.h>    unsigned **int** pow2(unsigned **int** **exp**) {  **if** (**exp** >= **sizeof**(unsigned **int**) \* CHAR\_BIT) {  /\* Handle error \*/  }  **return** 1 << **exp**;  } |

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

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

| **Principles(s):** **Validate input Data (1), Heed Compiler Warnings (2), Architect and Design for Security Policies (3), and Adopt a Secure Coding Standard (10):**  Integrate thorough integer precision checks for secure data validation.  Promptly address any compiler warnings related to integer precision for enhanced code reliability.  Embed comprehensive guidelines for secure integer handling into the architectural and design phase.  Expand our secure coding standards to encompass detailed rules on integer precision usage. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | N/A | Supported: Astree reports overflows due to insufficient precision. |
| CodeSonar | 7.4p0 | LANG.ARITH.BIGSHIFT | Shift Amount Exceeds Bit Width |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT35-a | Use correct integer precisions when checking the right hand operand of the shift operator |
| Polyspace Bug Finder | R2-23A | CERT C: Rule INT35-C | Checks for situations when integer precisions are exceeded (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Protection** | STD-010-JAV | Do not expose private members of an outer class from within a nested class  Ensuring that private members within a nested class stay secure is a fundamental principle of encapsulation and information hiding when practicing object-oriented programming principles. Following best practices will reduce any unintended access or modifications from external entities that should not have access to the private class members. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example exposes the private (x,y) coordinates through the getPoint() method of the inner class. Consequently, the AnotherClass class that belongs to the same package can also access the coordinates. |
| **class** Coordinates {  **private** **int** x;  **private** **int** y;  **public** **class** Point {  **public** **void** getPoint() {  System.out.println("(" + x + "," + y + ")");  }  }  }  **class** AnotherClass {  **public** **static** **void** main(String[] args) {  Coordinates c = **new** Coordinates();  Coordinates.Point p = c.**new** Point();  p.getPoint();  }  } |

| **Compliant Code** |
| --- |
| Use the private access specifier to hide the inner class and all contained methods and constructors. |
| **class** Coordinates {  **private** **int** x;  **private** **int** y;  **private** **class** Point {  **private** **void** getPoint() {  System.out.println("(" + x + "," + y + ")");  }  }  }  **class** AnotherClass {  **public** **static** **void** main(String[] args) {  Coordinates c = **new** Coordinates();  Coordinates.Point p = c.**new** Point(); // Fails to compile  p.getPoint();  }  } |

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

| **Principles(s):** **Heed Compiler Warnings (2), Architect and Design for Security Policies (3), Default Deny (5), and Adopt a Secure Coding Standard (10):**  Pay attention to any potential compiler warnings about exposing private members and promptly address them for code quality and security.  In the architectural and design phase, enforce policies that prevent the exposure of private members from any outer classes within a nested class, promoting security.  Implement a default deny strategy, which will restrict access to private members by default, further enhancing security measures.  Include guidelines for our developers in the coding standard that emphasize the importance of not exposing private members inside a nested class, ensuring we stay consistent, compliant, and secure. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | JAVA.CLASS. ICSBS | Inner Class Should be Static (Java) |
| Parasoft Jtest | 2023.1 | CERT.OBJ08. INNER | Make all member classes “private” |

### 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.



The primary purpose of automation is to help keep a program secure and compliant with constantly evolving coding standards. For instance, during the initial design phases, utilizing coding standards will help to eliminate vulnerabilities found during or after development. After this, during the building phase, the compilers will usually warn of critical errors in the code, allowing developers to correct any apparent errors at the time. After the program is built, automated static analyzers can be used to do a thorough and deep analysis of the code, finding any defects early in the code, saving time and money later, and making the existing code compliant and more secure. Lastly, after deployment, automation can constantly be used to help maintain the program and ensure it stays updated and secure and follows best practices and current coding standards.

### 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-CLG | Low | Unlikely | High | Low | 3 |
| STD-002-CPP | Medium | Unlikely | Medium | Medium | 3 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-JAV | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 5 |
| STD-006-CLG | Low | Unlikely | High | Low | 1 |
| STD-007-CPP | Low | Likely | Low | Low | 2 |
| STD-008-CPP | High | Likely | Medium | High | 5 |
| STD-009-CLG | Low | Unlikely | Medium | Low | 2 |
| STD-010-JAV | Medium | Likely | Medium | Medium | 3 |

### 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 | This encryption aims to safeguard data while it is stored on devices, databases, or any form of storage. In-rest encryption should be implemented where the data is stored to protect and prevent any unauthorized access during a breach or attack. This encryption ensures data confidentiality even when it's not actively used, preventing exposure. |
| Encryption at flight | This encryption aims to secure data during transmission over networks, like the internet or any internal communication channels. In-flight encryption should be implemented when sensitive data is transmitted between systems or over an untrusted network. This encryption prevents unauthorized access or eavesdropping, ensuring that the data stays confidential while in transit. |
| Encryption in use | This encryption protects data during active processing while it's used by applications or during any computation. This encryption can be implemented when the data is actively being used or processed to prevent any unauthorized access during use. This encryption ensures that any sensitive or personal information stays protected throughout its lifecycle. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is used to verify the identity of users, systems, or anyone attempting to access the network or application. This policy can be used to enforce that all systems and applications require access controls, preventing unauthorized entry. In addition, authentication is used to ensure that only authorized individuals or systems can access any sensitive data. |
| Authorization | Authorization controls and specifies the access level granted to authenticated users or systems. Authorization will be implemented in systems and applications to help restrict user permissions based on a role system. In addition, authorization can be implemented to prevent unauthorized access to sensitive information and maintain the principle of least privilege. |
| Accounting | Accounting will track and log the activities of users or systems to help establish responsibility for actions taken. Accounting can be applied in scenarios that can be traced and audited, especially in regulated industries. In addition, accounting will ensure security incidents are detected early and responded to and ensure we meet any regulatory and compliance requirements. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 11/12/2023 | Principles & Standards done | Tanner Holbrook | Tanner Holbrook |
| 3.0 | 12/3/2023 | Finished Document | Tanner Holbrook | Tanner Holbrook |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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