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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data is a key security measure that ensures only the correct and expected information is processed by the system. By checking the format and content of the data, developers can prevent attacks like SQL injection, cross-site scripting (XSS), and buffer overflows. Input validation acts as a filter, blocking any harmful or malformed data from reaching the application’s core processes, which ultimately strengthens security. |
| 1. Heed Compiler Warnings | Compiler warnings are an early sign of potential problems in code, and ignoring them can lead to serious security risks. Addressing these warnings can help catch issues such as memory leaks or logic errors before they become bigger vulnerabilities. By paying attention to these warnings, developers can maintain higher code quality and reduce the chance of security flaws down the line. |
| 1. Architect and Design for Security Policies | Incorporating security policies into the architecture and design of a system from the beginning is essential for building secure software. Security features like access control, encryption, and monitoring should be integrated into the system at the design phase. This proactive approach ensures that security is a core part of the system, reducing the likelihood of vulnerabilities being introduced later on. |
| 1. Keep It Simple | Simple designs are easier to understand, secure, and maintain. Complex systems often introduce unnecessary risk by increasing the likelihood of security vulnerabilities and making it harder to find and fix issues. By keeping systems and code as simple as possible, developers can reduce the attack surface and create software that is more reliable and secure. |
| 1. Default Deny | The default deny principle means restricting access by default and only granting it when absolutely necessary. This approach helps prevent unauthorized access by ensuring that users and processes only have access to resources when explicitly permitted. Default deny minimizes security risks by forcing developers to carefully consider which permissions are necessary for each user or system. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege limits the permissions and access granted to users and systems to only what is required for their tasks. By reducing unnecessary access, this principle lowers the potential impact of a security breach. If an account with minimal permissions is compromised, the attacker has less access to sensitive information or critical systems, making the damage much more contained. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to other systems, it is important to sanitize it to prevent potential security threats. By cleaning the data of any harmful elements, such as malicious code, developers can reduce the risk of injection attacks and buffer overflow exploits. Proper data sanitization ensures that the receiving system remains secure, maintaining the integrity of the communication between systems. |
| 1. Practice Defense in Depth | Defense in depth is a strategy that involves using multiple layers of security to protect a system. Even if one layer fails, other layers continue to provide protection, making it harder for an attacker to fully compromise the system. This layered approach is effective because it addresses security from different angles, using tools like firewalls, encryption, and monitoring systems to create a stronger defense. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance (QA) techniques, such as code reviews and security testing, are essential for catching vulnerabilities early in the development process. By thoroughly testing code and looking for weaknesses, developers can ensure that their software is secure before it’s deployed. Implementing strong QA practices helps improve both the security and overall quality of the software. |
| 1. Adopt a Secure Coding Standard | Secure coding standards provide guidelines for writing safe and reliable code that protects against common vulnerabilities. These standards help developers avoid mistakes like buffer overflows or injection flaws by following proven best practices. By adopting a secure coding standard, teams can create more consistent and secure software, reducing the chances of introducing security issues during development. |

### 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** | **Implement abstract data types using opaque types** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | This approach hides the implementation details of data structures, promoting encapsulation and reducing the risk of unintended interference with the internal state, thereby enhancing code safety and maintainability. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is based on the managed string library developed by CERT [[Burch 2006](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Burch06)]. In this example, the managed string type and the functions that operate on this type are defined in the string\_m.h header file as follows: |
| 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 compliant solution makes the string\_mx type private, concealing its implementation from users of the managed string library. The developer achieves this by creating two header files: a public string\_m.h for users and an internal file for implementing the abstract data type. |
| 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);  /\* ... \*/ |

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

| **Principles(s):** Default Deny: This prevents unauthorized access or modification by limiting visibility and access to only what is explicitly allowed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL12** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **104 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2021.1 | **CERT\_C-DCL12-a** | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2021a | [CERT C: Rec. DCL12-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.dcl12c.html) | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | When converting between different integer types, it’s crucial to prevent data loss or inaccuracies that can lead to logical errors or unexpected behavior in the program, which could compromise its functionality. |

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

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

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

|  |
| --- |
| **Principles(s):** ValidateInput Data: This ensures that all input data is validated to avoid unintended behavior, such as unsigned integer wrapping, which could lead to data loss or misinterpretation during conversions. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 20.10 | integer-overflow | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=125337650) | 7.2.0 | CertC-INT30 | Implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2021.1 | CERT\_C-INT30-a CERT\_C-INT30-b CERT\_C-INT30-c | Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2021a | [CERT C: Rule INT30-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint30c.html) | Checks for:  Unsigned integer overflow  Unsigned integer constant overflow  Rule partially covered. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Use valid references, pointers, and iterators to reference elements of a basic\_string** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Utilizing valid references, pointers, and iterators ensures safe access to string elements, preventing dereferencing of null or invalid addresses that can lead to segmentation faults or undefined behavior. This practice promotes safer, more reliable code by ensuring that all memory access is legitimate and within bounds. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example copies input into a std::string, replacing semicolon (;) characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard: Adhering to such standards helps prevent improper use of references, pointers, or iterators that could lead to unsafe modifications or undefined behavior with string elements. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 157 S | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2021.1 | CERT\_C-STR30-a CERT\_C-STR30-b | A string literal shall not be modified Do not modify string literals |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | 489, 1776 | Partially supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2021a | [CERT C: Rule STR30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr30c.html) | Checks for writing to const qualified object (rule fully covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection attacks exploit vulnerabilities in application code to manipulate database queries. By validating inputs and using prepared statements, we can safeguard against these attacks and protect sensitive data. |

| **Noncompliant Code** |
| --- |
| The JDBC library’s PreparedStatement class sanitizes untrusted data, preventing SQL injection when used correctly. However, this example modifies doPrivilegedAction() to use a PreparedStatement, but still risks SQL injection by directly incorporating the unsanitized username input. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;    class Login {    public Connection getConnection() throws SQLException {      DriverManager.registerDriver(new              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"      return DriverManager.getConnection(dbConnection);    }      String hashPassword(char[] password) {      // Create hash of password    }      public void doPrivilegedAction(      String username, char[] password    ) throws SQLException {      Connection connection = getConnection();      if (connection == null) {        // Handle error      }      try {        String pwd = hashPassword(password);        String sqlString = "select \* from db\_user where username=" +          username + " and password =" + pwd;        PreparedStatement stmt = connection.prepareStatement(sqlString);          ResultSet rs = stmt.executeQuery();        if (!rs.next()) {          throw new SecurityException("User name or password incorrect");        }          // Authenticated; proceed      } finally {        try {          connection.close();        } catch (SQLException x) {          // Forward to handler        }      }    }  } |

| **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):** Keep It Simple – when not explicitly constructing and destructing manually managed objects, complexity is greatly increased. Explicit statements will simplify the process and prevent errors from occurring. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2021.1 | C++4761, C++4762, C++4766, C++4767 |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.1 | CERT\_CPP-MEM53-a | Do not invoke malloc/realloc for objects having constructors |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.07 | [V749](https://www.viva64.com/en/w/v749/) |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Accessing memory after it has been deallocated can lead to unpredictable behavior, including crashes or data corruption. Properly managing memory helps ensure the stability and reliability of the application. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, s is dereferenced after being deallocated, risking a write-after-free exploit to execute arbitrary code. The function g() is marked noexcept(false) to comply with MEM52-CPP, detecting and handling memory allocation errors. |
| #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):** Keep It Simple: Using clear and explicit statements, such as parameterized queries, helps simplify the process of interacting with databases and reduces the risk of introducing vulnerabilities like SQL injection, which can occur with complex, dynamically constructed queries. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2021.1 | C++4761, C++4762, C++4766, C++4767 |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.1 | CERT\_CPP-MEM53-a | Do not invoke malloc/realloc for objects having constructors |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.07 | [V749](https://www.viva64.com/en/w/v749/) |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Incorporate diagnostic tests using assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions help catch programming errors during development by validating assumptions in the code. This proactive approach facilitates debugging and enhances overall code quality. |

| **Noncompliant Code** |
| --- |
| This noncompliant example uses assert() to verify memory allocation, which can lead to abrupt termination if memory exhaustion occurs. A robust program must handle such failures gracefully, as relying on assert() may result in a denial-of-service vulnerability. |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char \*)malloc(len + 1);    assert(NULL != dup);      memcpy(dup, c\_str, len + 1);    return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char\*)malloc(len + 1);    /\* Detect and handle memory allocation error \*/    if (NULL == dup) {        return NULL;    }    memcpy(dup, c\_str, len + 1);    return dup;  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques: Incorporating assertions into your code helps detect issues during development, ensuring that software behaves as expected, which is essential for maintaining quality and reliability. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Tool | Version | Checker | Description |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Can detect some violations of this rule. However, it can only detect violations involving abort() because assert() is implemented as a macro |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 44 S | Enhanced enforcement |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2021.1 | CERT\_C-ERR06-a | Do not use assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception handling ensures that the program can gracefully recover from unexpected errors, improving user experience and preventing crashes that could lead to data loss or security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings: Properly handling exceptions ensures that the compiler can safely manage stack unwinding and other critical operations, preventing runtime errors and improving the robustness of the code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 527 S | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.1 | CERT\_CPP-ERR51-a CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2020a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [PRQA QA-C++](https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=142409849) | 4.4 | 4035, 4036, 4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not alternately input and output from a file stream without an intervening positioning call** |
| --- | --- | --- |
| Input and Ouput | [STD-008-CPP] | Switching between reading and writing without repositioning can lead to data being written in the wrong location or read incorrectly. Ensuring proper positioning maintains data integrity. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example appends data to the end of a file and then reads from the same file. However, because there is no intervening positioning call between the formatted output and input calls, the behavior is undefined. |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }      file << "Output some data";    std::string str;    file >> str;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::basic\_istream<T>::seekg() function is called between the output and input, eliminating the undefined behavior. |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    file << "Output some data";      std::string str;    file.seekg(0, std::ios::beg);    file >> str;  } |

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

| **Principles(s):** Keep It Simple: When alternating between input and output operations in a file stream without an intervening positioning call (e.g., fseek or rewind), it can lead to unexpected behavior, data corruption, or security vulnerabilities. Keeping operations simple and well-ordered (by using positioning calls) prevents confusion and unexpected results. Simplicity in design avoids edge cases that could be exploited. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-FIO50** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **IO.IOWOP**  **IO.OIWOP** | Input After Output Without Positioning  Output After Input Without Positioning |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **DF4711, DF4712, DF4713** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-FIO50-a** | Do not alternately input and output from a stream without an intervening flush or positioning call |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: FIO50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio50cpp.html) | Checks for alternating input and output from a stream without flush or positioning call (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Guarantee that container indices and iterators are within the valid range** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Accessing out-of-bounds elements can result in undefined behavior or crashes. Validating indices and iterators helps maintain program stability and prevents potential vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates a function, insert\_in\_table(), with two int parameters, pos and value, influenced by untrusted data. While it checks the upper bound against tableSize, it neglects the lower bound, allowing pos (as a signed int) to take negative values, which can lead to memory write errors. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

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

| **Principles(s):** Validate Input Data: This prevents out-of-bounds errors, which can lead to memory corruption, crashes, or even vulnerabilities like buffer overflows. Validating the range protects the integrity of the system by preventing unauthorized access to memory areas. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **overflow\_upon\_dereference** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.MEM.BO LANG.MEM.BU LANG.MEM.TO LANG.MEM.TU LANG.MEM.TBA LANG.STRUCT.PBB LANG.STRUCT.PPE LANG.STRUCT.PARITH** | Buffer overrun Buffer underrun Type overrun Type underrun Tainted buffer access Pointer before beginning of object Pointer past end of object Pointer Arithmetic |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **C++3139, C++3140**  **DF2891** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.2 | **ABV.ANY\_SIZE\_ARRAY** **ABV.GENERAL** **ABV.GENERAL.MULTIDIMENSION** **ABV.STACK** **ABV.TAINTED** **SV.TAINTED.ALLOC\_SIZE** **SV.TAINTED.CALL.INDEX\_ACCESS** **SV.TAINTED.CALL.LOOP\_BOUND** **SV.TAINTED.INDEX\_ACCESS** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-CTR50-a** | Guarantee that container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: CTR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr50cpp.html) | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.33 | [**V781**](https://pvs-studio.com/en/docs/warnings/v781/) |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| Resources | [STD-010-CPP] | Failing to free dynamically allocated memory leads to memory leaks, which can exhaust system resources over time. Proper deallocation is crucial for efficient resource management and application performance. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the local variable space is passed as the expression to the placement new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined behavior due to ::operator delete() attempting to free memory that was not returned by ::operator new(). |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      delete s1;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. This is one of the few times when explicitly invoking a destructor is warranted. |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      s1->~S();  } |

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

| **Principles(s):** Adopt a Secure Coding Standard: Improper resource management leads to memory leaks or worse, potential security vulnerabilities such as heap exploitation. Secure coding standards ensure that best practices like memory management are consistently followed, preventing resource exhaustion attacks or unintended behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **invalid\_dynamic\_memory\_allocation dangling\_pointer\_use** |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MEM51** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.2 | **C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Klocwork) | 2024.2 | **CL.FFM.ASSIGN** **CL.FFM.COPY** **CL.FMM** **CL.SHALLOW.ASSIGN** **CL.SHALLOW.COPY** **FMM.MIGHT** **FMM.MUST** **FNH.MIGHT** **FNH.MUST** **FUM.GEN.MIGHT** **FUM.GEN.MUST** **UNINIT.CTOR.MIGHT** **UNINIT.CTOR.MUST** **UNINIT.HEAP.MIGHT** **UNINIT.HEAP.MUST** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MEM51-a** **CERT\_CPP-MEM51-b** **CERT\_CPP-MEM51-c** **CERT\_CPP-MEM51-d** | Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources |

### 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.

Green Pace’s established DevOps process provides a solid foundation to introduce automated security measures at key stages. The DevSecOps approach integrates security throughout the DevOps pipeline, transforming it into a more secure and efficient process.

In the pre-production stages, automation can be introduced during the “Assess and Plan” phase for threat modeling, and in the “Design” and “Build” phases, where IDE security is prioritized. Automated unit and static application testing, along with security scans, can be incorporated into the “Verify & Test” phase to ensure code quality and compliance before deployment.

Once in production, automation can continue to enhance security through penetration testing, integrity checks, and defense-in-depth measures. Monitoring tools can automatically alert teams of risky events or performance issues, enabling full-time coverage and rapid response to threats. These automated processes ensure continuous compliance and a proactive approach to security, from development through deployment.

### 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 | Low | Unlikely | High | P1 | L3 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | Low | Likely | Low | P9 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Likely | Medium | P6 | L2 |
| STD-009-CPP | High | Likely | High | P9 | L2 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | This policy ensures that all sensitive data stored on physical devices or cloud storage is protected by encryption, preventing unauthorized access even if someone gains physical access to the storage. This includes hard drives, mobile devices, computers, and cloud assets, where encryption tools such as disk encryption and mobile security protocols safeguard the data. By requiring encryption keys to access the information, the policy protects against internal and external threats, ensuring that sensitive data remains secure and unusable without proper authorization. |
| Encryption in flight | This ensures that data is encrypted while being transmitted between devices, whether within a network or externally. This protection is crucial because data in transit is vulnerable to interception through insecure connections, such as unprotected Wi-Fi or cloud networks. Encrypting data during transmission prevents attackers from accessing sensitive information mid-transfer. Methods like email encryption, Data Loss Prevention (DLP) solutions, firewalls, and strong authentication measures are essential to safeguarding the data as it moves, ensuring that only authorized recipients can access it. |
| Encryption in use | This ensures that sensitive data remains encrypted while it is being actively accessed, created, or modified. This policy is important because it prevents data from being exposed during moments of vulnerability, such as when it is being processed in memory or actively used by applications. By applying encryption at all stages, it ensures continuous protection through defense-in-depth, layering multiple defenses around the data to minimize potential attack points. Managing access rights and identity is also key to limiting exposure, ensuring only authorized users or processes can interact with the data while it is in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This ensures that a user's identity is verified before granting access to sensitive systems or data. This process involves various methods, such as static passwords, one-time passwords, certifications, and biometric credentials, all designed to confirm that the individual is who they claim to be. Authentication acts as a critical first layer of defense, allowing the system to either accept or reject access based on established criteria. This policy is essential to protect sensitive information and applies to all users seeking access, ensuring that only authorized individuals can interact with the system. |
| Authorization | This defines the rights and privileges an authenticated user has, determining what they can and cannot access within a system. While authentication confirms a user's identity, authorization limits their access to only the resources necessary to perform their role. By applying the principle of least privilege, this policy minimizes potential security risks, as users are restricted from accessing sensitive data or performing actions beyond their required duties. This helps prevent unauthorized activities and reduces vulnerabilities in the system. |
| Accounting | This involves tracking and recording user activities within a system, such as timestamps, accessed resources, and data transfers. This process creates an audit trail that can be invaluable for compliance, security, and forensic analysis. By monitoring interactions, the policy helps identify risky behavior early, ensuring that potential threats can be addressed before significant damage occurs. It also holds users accountable for their actions and can provide evidence in case of a security breach, helping to pinpoint system vulnerabilities or errors that may have been exploited. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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