**Green Pace Developer: Security Policy Guide**



# 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 | Always check and verify all input from users or external sources before processing it. Invalid or malicious input can cause vulnerabilities like injection attacks, buffer overflows, or unexpected behavior. Strong input validation ensures data conforms to expected formats and types, reducing the risk of security breaches. |
| 1. Heed Compiler Warnings | Treat compiler and static analysis warnings seriously, as they often highlight potential security flaws or code that could behave unpredictably. Addressing these warnings early helps catch bugs and vulnerabilities before they reach production, promoting more robust and secure software. |
| 1. Architect and Design for Security Policies | Incorporate security requirements and controls at the earliest stages of system architecture and design. Anticipate threats and define clear security policies (authentication, authorization, and data protection) so they are enforced throughout the system’s lifecycle, not retrofitted later. |
| 1. Keep It Simple | Favor simple, well-understood designs and implementations. Complexity increases the chance of errors, makes code harder to audit, and often introduces subtle bugs that can be exploited. Simpler systems are easier to secure, test, and maintain. |
| 1. Default Deny | Configure systems and applications to deny access by default, only granting permissions explicitly as needed. This minimizes the attack surface by ensuring no user or process has more rights than necessary unless specifically authorized. |
| 1. Adhere to the Principle of Least Privilege | Grant each user, process, or system component only the minimum access or permissions needed to perform its function (Role Based Auth). Limiting privileges reduces potential damage if an account or process is compromised and helps contain security breaches. |
| 1. Sanitize Data Sent to Other Systems | Before sharing data with other programs, components, or services, cleanse it of any potentially harmful content. Non-sanitized data can be used to exploit vulnerabilities in downstream systems, so rigorous sanitization is crucial for preventing attacks like command injection or cross-site scripting. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls and safeguards throughout the system. If one layer fails, others still provide protection. This layered approach increases overall resilience and makes successful attacks much harder. |
| 1. Use Effective Quality Assurance Techniques | Apply comprehensive quality assurance methods such as code reviews, automated testing, and static/dynamic analysis to find and fix security weaknesses. Early and continuous testing reduces the risk of vulnerabilities making it to production. |
| 1. Adopt a Secure Coding Standard | Follow a well-established secure coding standard relevant to your language and environment. Consistently applying proven guidelines helps developers avoid common pitfalls, leading to more secure, reliable, and maintainable code. |

### 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** | Do not cast to an out-of-range enumeration value. |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Avoid casting an integer that is not a valid enum value, otherwise, the resulting enum is unpredicatable and may cause errors, and undefined behavior.  Check the integer’s range before casting to an enum to ensure everything is correct. |

| **Noncompliant Code** |
| --- |
| In this case, the code casts an integer to an enum, but without first checking if it is a valid value. This can create undefined behavior and errors. |
| 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 prior to performing the conversion, which guarantees the conversion does not result in an unspecified value. It accomplishes 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); } |

| **Principles(s):** Validate Input Data ensures the input integer is checked before casting, preventing undefined behavior. Architect and Design for Security Policies embeds type safety into design. Adopt a Secure Coding Standard reinforces secure practices as per SEI CERT. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S131 | Checks enum boundary casting. |

#### Coding Standard 2

| **Coding Standard** | **Label** | Avoid information leakage when passing a class object across a trust boundary |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Padding bytes or bits inside objects may contain sensitive data or not. If copying objects across boundries, such as other software, processes, users, remember to serialize or zero objects before transfer. |

| **Noncompliant Code** |
| --- |
| This noncompliant code runs in a kernel space, copying data from arg to the user space. Though, padding bits may be utilized by objects internally, possibly carrying sensitive information. This can lead to data leaks no matter how the data is copied. |
| #include <cstddef>   struct test {  int a;  char b;  int c; };   // Safely copy bytes to user space extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);   void do\_stuff(void \*usr\_buf) {  test arg{1, 2, 3};  copy\_to\_user(usr\_buf, &arg, sizeof(arg)); } |

| **Compliant Code** |
| --- |
| This compliant solution serializes the structure data before copying it to an untrusted context. This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the copy\_to\_user() function would need to unpack it to recreate the original, padded structure. |
| #include <cstddef> #include <cstring>   struct test {  int a;  char b;  int c; };   // Safely copy bytes to user space. extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);   void do\_stuff(void \*usr\_buf) {  test arg{1, 2, 3};  // May be larger than strictly needed.  unsigned char buf[sizeof(arg)];  std::size\_t offset = 0;    std::memcpy(buf + offset, &arg.a, sizeof(arg.a));  offset += sizeof(arg.a);  std::memcpy(buf + offset, &arg.b, sizeof(arg.b));  offset += sizeof(arg.b);  std::memcpy(buf + offset, &arg.c, sizeof(arg.c));  offset += sizeof(arg.c);    copy\_to\_user(usr\_buf, buf, offset /\* size of info copied \*/); } |

| **Principles(s):** Sanitize Data Sent to Other Systems ensures sensitive data does not leak across boundaries. Default Deny prevents unauthorized data exposure. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.12 | Information Leak | |  | | --- | |  |  |  | | --- | | Detects potential leakage of padding bits. | |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow.  Buffer overflows occur frequently when manipulating strings. To prevent such errors, either limit copies through truncation or ensure that the destination is of sufficient size to hold the data to be copied. |

| **Noncompliant Code** |
| --- |
| In this example, the code does not allocate enough memory for the null character. This may pass initially, though after the first buffer is executed, the next one will overflow. |
| #include <iostream>   void f() {  char bufOne[12];  char bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo; } |

| **Compliant Code** |
| --- |
| The compliant fix is to use std::string instead, making sure that the data is not truncated out of range. |
| #include <iostream> #include <string>   void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo; } |

| **Principles(s):** Validate Input Data ensures adequate buffer size. Keep It Simple reduces complexity, using safe constructs like std::string. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S5782 | Prevents buffer overflow. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | STD-004-JAV | SQL injection vulnerabilities come about in applications where elements of a SQL query originate from an untrusted source. This is usually due to altering the query, causing information leak or even data alterations.  The primary means of preventing SQL injection are sanitization and validation. This can be implemented as parameterized queries and stored procedures. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a bad actor would be able to alter the SQL query. This would erroneously allow for data alterations or private data to be wrangled by the bad actor. |
| 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  }  }  }  }  delete pst; } |

| **Compliant Code** |
| --- |
| By using the set\*() methods of the PreparedStatement class, one may enforce strong type checking.  This technique mitigates the SQL injection vulnerability because the input has properly escaped.  Note that prepared statements must be used even with queries that insert data into the database. |
| 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  }  }  } |

| **Principles(s):** Sanitize Data Sent to Other Systems ensures SQL queries are sanitized. Default Deny restricts unauthorized query alterations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.14.0 | SQL Injection | Automated SQL injection testing. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Explicitly construct and destruct objects when manually managing object lifetime** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | When manually managing the lifetime of an object, the constructor must be called during instantiation.  Also, the destructor must be called to terminate the lifetime of the object. Use of an object outside of its lifetime is undefined behavior.  An object can be constructed either by calling the constructor using the placement new operator or by calling the construct() function of an allocator object. Much the same, an object can be destroyed either by calling the destructor explicitly or by calling the destroy() function of an allocator object. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a custom container class uses an allocator object to obtain storage for arbitrary element types.  While the copy\_elements() function is considered to call copy constructors for elements being moved into the newly allocated storage, this example does not call the default constructor for any additional elements being allocated.  When an element is accessed through the operator function, it results in undefined behavior, depending on the type T. |
| #include <memory>  template <typename T, typename Alloc = std::allocator<T>> class Container {  T \*underlyingStorage;  size\_t numElements;    void copy\_elements(T \*from, T \*to, size\_t count);   public:  void reserve(size\_t count) {  if (count > numElements) {  Alloc alloc;  T \*p = alloc.allocate(count); // Throws on failure  try {  copy\_elements(underlyingStorage, p, numElements);  } catch (...) {  alloc.deallocate(p, count);  throw;  }  underlyingStorage = p;  }  numElements = count;  }    T &operator[](size\_t idx) { return underlyingStorage[idx]; }  const T &operator[](size\_t idx) const { return underlyingStorage[idx]; } }; |

| **Compliant Code** |
| --- |
| In this compliant solution, all elements are properly initialized by explicitly calling copy or default constructors for T. |
| #include <memory>  template <typename T, typename Alloc = std::allocator<T>> class Container {  T \*underlyingStorage;  size\_t numElements;    void copy\_elements(T \*from, T \*to, size\_t count);   public:  void reserve(size\_t count) {  if (count > numElements) {  Alloc alloc;  T \*p = alloc.allocate(count); // Throws on failure  try {  copy\_elements(underlyingStorage, p, numElements);  for (size\_t i = numElements; i < count; ++i) {  alloc.construct(&p[i]);  }  } catch (...) {  alloc.deallocate(p, count);  throw;  }  underlyingStorage = p;  }  numElements = count;  }    T &operator[](size\_t idx) { return underlyingStorage[idx]; }  const T &operator[](size\_t idx) const { return underlyingStorage[idx]; } }; |

| **Principles(s):** Architect and Design for Security ensures correct lifecycle management. Adopt a Secure Coding Standard advocates proper resource management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.12 | Resource Leak | Ensures proper allocation/deallocation. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Value-returning functions must return a value from all exit paths** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, like a function-try-block.    Flowing off the end of a function-try-block is equivalent to a return with no value. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function-try-block handler does not return a value, resulting in undefined behavior when an exception is thrown. |
| #include <vector>   std::size\_t f(std::vector<int> &v, std::size\_t s) try {  v.resize(s);  return s; } catch (...) { } |

| **Compliant Code** |
| --- |
| In this compliant solution, the exception handler of the function-try-block also returns a value. |
| #include <vector>   std::size\_t f(std::vector<int> &v, std::size\_t s) try {  v.resize(s);  return s; } catch (...) {  return 0; } |

| **Principles(s):** Validate Input Data ensures expected return values. Use Effective Quality Assurance Techniques ensures thorough testing of code paths. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | High | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S935 | Checks all return paths. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown.  If no matching handler is found within the handlers for a try-block, the handler searches proceeds; it investigates the surrounding try blocks of the same thread.  Exceptions thrown inside the thread that are not caught, cause the thread to call std::terminate() and possibly crash the process. |

| **Noncompliant Code** |
| --- |
| For this noncompliant code sample, the thread entry point function thread\_start() does not catch exceptions thrown by throwing\_func().  If the initial thread function exits because an exception is thrown, std::terminate() will be called. |
| #include <thread>  void throwing\_func() noexcept(false);   void thread\_start() {  throwing\_func(); }   void f() {  std::thread t(thread\_start);  t.join(); } |

| **Compliant Code** |
| --- |
| In the compliant solution, the thread\_start() handles all exceptions and does not rethrow, which allows the thread to terminate normally. |
| #include <thread>  void throwing\_func() noexcept(false);  void thread\_start(void) {  try {  throwing\_func();  } catch (...) {  // Handle error  } }  void f() {  std::thread t(thread\_start);  t.join(); } |

| **Principles(s):** Practice Defense in Depth ensures robust error handling. Use Effective Quality Assurance Techniques prevents crashes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.12 | Uncaught Exceptions | Detects uncaught exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Prefer special member functions and overloaded operators to C Standard Library functions** |
| --- | --- | --- |
| Object-Oriented Programming | STD-008-CPP | Several C standard library functions perform bytewise operations on objects.    Functions like memcpy or memcmp only copy bytes, not object semantics. For objects with pointers, destructors, or special logic, using these functions leads to shallow copies, double-free bugs, or corruption. Use copy constructors and assignment operators instead. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, std::memcpy() is used, creating a copy of an object of nontrivial type C.  Though, each object instance try's to delete the int \* in the destructor, double-free vulnerabilities can occur because the same pointer value will be copied into c2. |
| #include <cstring>   class C {  int \*i;   public:  C() : i(nullptr) {}  ~C() { delete i; }    void set(int val) {  if (i) { delete i; }  i = new int{val};  }    // ... };   void f(C &c1) {  C c2;  std::memcpy(&c2, &c1, sizeof(C));  } |

| **Compliant Code** |
| --- |
| In this compliant solution, C defines an assignment operator that is used instead of calling std::memcpy(). This makes sure the object is copied according to its bounds. |
| class C {  int \*i;   public:  C() : i(nullptr) {}  ~C() { delete i; }    void set(int val) {  if (i) { delete i; }  i = new int{val};  }   C &operator=(const C &rhs) noexcept(false) {  if (this != &rhs) {  int \*o = nullptr;  if (rhs.i) {  o = new int;  \*o = \*rhs.i;  }  // Does not modify this unless allocation succeeds.  delete i;  i = o;  }  return \*this;  }    // ... };   void f(C &c1) {  C c2 = c1; } |

| **Principles(s):** Keep It Simple and Architect and Design for Security Policies reinforce object-oriented security practices, preventing subtle vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S6052 | Discourages unsafe standard C functions. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Predicate function objects should not be mutable** |
| --- | --- | --- |
| Containers | STD-009-CPP | Processes may copy predicates, and if it tracks state (such as a counter), that state may fall out of sync. Mutable predicates can cause non-deterministic bugs. Keep predicates stateless or use external state tracking. |

| **Noncompliant Code** |
| --- |
| With this noncompliant code example, the code attempts to remove the third item in a container using a predicate lambda function that returns true only on its third invocation.  Although, this code uses a mutable predicate, allowing algorithms to copy it, causing non determined behavior and desync. |
| #include <algorithm> #include <iostream> #include <iterator> #include <vector>   template <typename Iter> void print\_container(Iter b, Iter e) {  std::cout << "Contains: ";  std::copy(b, e, std::ostream\_iterator<decltype(\*b)>(std::cout, " "));  std::cout << std::endl; }   void f() {  std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  print\_container(v.begin(), v.end());   int timesCalled = 0;  v.erase(std::remove\_if(v.begin(), v.end(), [timesCalled](const int &) mutable { return ++timesCalled == 3; }), v.end());  print\_container(v.begin(), v.end()); } |

| **Compliant Code** |
| --- |
| By avoiding a mutable predicate altogether, the code results in predictable behavior, no matter how the predicate is copied. |
| #include <algorithm>  #include <iostream>  #include <iterator>  #include <vector>    template <typename Iter>  void print\_container(Iter B, Iter E) {  std::cout << "Contains: ";  std::copy(B, E, std::ostream\_iterator<decltype(\*B)>(std::cout, " "));  std::cout << std::endl;  }    void f() {  std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  print\_container(v.begin(), v.end());  v.erase(v.begin() + 3);  print\_container(v.begin(), v.end());  } |

| **Principles(s):** Architect and Design for Security Policies ensures predictable, stateless predicates. Keep It Simple avoids unpredictable behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 18.0 | bugprone-lambda-function-name | Identifies mutable lambda issues. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| File-Handling (Input-Output) | STD-010-CPP | A call to the std::basic\_filebuf::open() function must be coupled with a call to std::basic\_filebuf::close() prior to the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.  Open files consume resources and, if not closed, can leak handles or lock files, eventually exhausting system resources. Always close files promptly, ideally using RAII (Resource Acquisition Is Initialization) to guarantee cleanup. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example, it is assumed to call std::terminate() at some point.   Do not abruptly terminate the program, as this can leave the open file hanging, leaking resources or corrupting the file altogether. |
| #include <exception> #include <fstream> #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate(); } |

| **Compliant Code** |
| --- |
| In this compliant solution, the stream is automatically closed through RAII before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception> #include <fstream> #include <string>  void f(const std::string &fileName) {  {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  } // file is closed properly here when it is destroyed  std::terminate(); } |

| **Principles(s):** Adhere to Least Privilege and Keep It Simple ensure resource leaks don't compromise stability and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S2095 | Checks file resource leaks. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



### Automation



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.

Automation plays a pivotal role in integrating secure coding practices directly into Green Pace's established DevSecOps workflow. Static application security testing (SAST) tools, such as SonarQube and Coverity, are utilized at the earliest stages—during the "Plan," "Create," and "Verify" phases—to immediately identify code vulnerabilities and enforce coding standards. By embedding these tools within the Integrated Development Environment (IDE) and build pipelines, vulnerabilities are detected in real-time, drastically reducing the likelihood of critical security issues reaching production.

Dynamic application security testing (DAST) tools, like OWASP ZAP, complement this by performing vulnerability scans during the "Pre-Production" phase, simulating attacks against running software to catch runtime flaws, injection attacks, and authentication issues. Input fuzzing and chaos engineering methods will also be employed in this phase to stress-test applications under realistic, unexpected scenarios.

In the "Release" phase, automated software signing processes and integrity checks ensure that only verified, secure builds are deployed. Further, real-time application self-protection (RASP) and user and entity behavior analytics (UEBA) tools will monitor live environments during "Detect" and "Respond" phases, rapidly identifying anomalous behavior and triggering automated security responses, such as alerts and isolation measures. Collectively, these automated practices reinforce a robust defense-in-depth posture, enhancing Green Pace's resilience against emerging threats.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Medium | Medium | High | 4 |
| STD-002-CPP | High | Medium | Medium | High | 4 |
| STD-003-CPP | High | High | Low | Critical | 5 |
| STD-004-JAV | Critical | High | Medium | Medium | 5 |
| STD-005-CPP | Medium | Medium | Medium | High | 3 |
| STD-006-CPP | Medium | Medium | Low | High | 4 |
| STD-007-CPP | High | Medium | Medium | Medium | 4 |
| STD-008-CPP | Medium | Medium | Low | Medium | 3 |
| STD-009-CPP | Medium | Medium | Low | Medium | 3 |
| STD-010-CPP | Medium | Medium | Low | Medium | 3 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the process of securing stored data, protecting sensitive information residing on disk drives, databases, backups, and other persistent storage solutions. At Green Pace, AES-256 encryption standards are mandated for all databases and stored files containing sensitive data. This encryption ensures that even in the event of unauthorized access or hardware theft, the stored data remains unreadable and secure. Implementation of this policy is mandatory for all data storage solutions and is continuously validated through automated compliance checks and regular audits. |
| Encryption in flight | Encryption in flight secures data transmitted over networks, preventing interception and tampering during communication between systems and users. Green Pace employs Transport Layer Security (TLS) version 1.3 as the minimum required protocol for all data in transit, ensuring robust confidentiality, integrity, and authentication. All internal and external communications—including API calls, user sessions, and data replication between services—must comply with this standard, validated by automated security scans and regular network traffic monitoring tools. |
| Encryption in use | Encryption in use pertains to protecting sensitive data actively being processed in memory or within computational tasks. Green Pace utilizes confidential computing technologies such as Intel SGX and secure memory enclaves to create isolated processing environments. These technologies ensure sensitive operations (like cryptographic key handling or processing personally identifiable information) are secure against threats that target memory directly, such as memory scraping or side-channel attacks. Encryption in use is required for handling highly sensitive computations and is integrated into Green Pace's architecture at the hardware and software layers, validated via regular security reviews. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies user identities to ensure only authorized individuals access Green Pace’s resources. Strong authentication mechanisms such as multi-factor authentication (MFA) combining knowledge factors (passwords), possession factors (security tokens), and inherence factors (biometrics) are enforced for all sensitive applications and administrative interfaces. User logins are securely logged and monitored for anomalous activities, ensuring accountability and rapid detection of potential credential misuse. |
| Authorization | Authorization policies define and enforce user privileges, ensuring each user or system process has access only to the resources necessary for their role. Green Pace strictly implements role-based access control (RBAC) models, clearly defining roles with associated permissions and regularly auditing user access levels. Changes to user privileges, additions of new users, and permissions granted to files or database objects are rigorously tracked, maintaining strict compliance with the principle of least privilege. This structured approach significantly mitigates the risk of unauthorized access or privilege escalation attacks. |
| Accounting | Accounting practices at Green Pace involve comprehensive monitoring, logging, and auditing of all user activities and system interactions. Detailed logs track user logins, changes to databases, the addition or modification of user accounts, access levels, and files accessed by users. Automated log analysis tools (such as Splunk or ELK Stack) systematically review these logs to identify suspicious activities, compliance violations, or potential breaches. Accounting provides Green Pace with critical visibility for forensic analysis, compliance verification, and continuous improvement of security practices. |

**\***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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/23/2025 | Defined security principles and secure coding standards | Zane Deso | Awt\_Approval |
| 1.2 | 6/20/2025 | Integrated security policy outlining principles, threat levels, and automation tools for security coding standards. Completed automation benefits, encryption policies, and triple-a policies. | Zane Deso | Awt\_Approval |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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