**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

# Victor Jarvis

## 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 | All input data must be properly checked and sanitized to prevent injection attacks and other malicious inputs. |
| 1. Heed Compiler Warnings | Any compiler warnings should be addressed as soon as possible to identify any potential vulnerabilities, especially early in the development process. |
| 1. Architect and Design for Security Policies | Integrate security policies into the architecture and design phases to build security into the system from the ground up, as well as set a standard from the beginning. |
| 1. Keep It Simple | Simplifying design and code to minimize potential security flaws is going to make it easier to identify and fix vulnerabilities. |
| 1. Default Deny | A default deny stance is where access to resources is denied by default and only explicitly granted permissions are allowed. This further prevents any unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Limiting user and system access rights to the minimum necessary to perform their tasks, reducing the potential damage from breaches. |
| 1. Sanitize Data Sent to Other Systems | Ensure that data is cleansed before it is transmitted to other systems to prevent the spread of malicious code or data corruption. |
| 1. Practice Defense in Depth | Implementing multiple layers of security controls provide redundancy and increase overall protection, making it harder for an attacker to breach the system. |
| 1. Use Effective Quality Assurance Techniques | Implementing rigorous quality assurance processes, including regular code reviews and testing, will help to identify and address security issues. |
| 1. Adopt a Secure Coding Standard | Following established secure coding standards will reduce vulnerabilities and promote consistency and security in code development. Integrating security measures into the system's architecture from the very beginning will ensure protection throughout the lifecycle of the system. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Obey the one definition rule** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | “Every program shall contain exactly one definition of every non-inline function or variable… used in that program”  Each entity should be defined only once in the program to avoid conflicts and inconsistencies. While there can be multiple declarations (statements that specify the type and name of a variable or function but do not allocate storage or provide an implementation), there should be only one definition that allocates storage or provides an implementation. |

| **Noncompliant Code** |
| --- |
| This example features two different units defining a class of the same name, but with different definitions. Even though they function the same way, they are not defined using the same token sequence. |
| // a.cpp  struct S {  int a;  };    // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| A header file can be used to introduce the object into both units. |
| // S.h  struct S {  int a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

| **Principles:**  **Heed Compiler Warnings**: This ensures consistent definitions across the code, avoiding undefined behavior that compilers would warn about.  **Keep It Simple**: Simplicity ensures that the code is clear and maintainable, reducing risk of errors.  **Adopt a Secure Coding Standard**: Following standards ensures consistency and prevents subtle bugs. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | type-compatibility  definition-duplicate  undefined-extern  undefined-extern-pure-virtual  external-file-spreading  type-file-spreading | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL60-a | A class, union or enum name (including qualification, if any) shall be a unique identifier |
| Polyspace Bug Finder | R2024a | CERT C++: DCL60-CPP | Checks for inline constraints not respected (rule partially covered) |

#### 

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not rely on the value of a moved-from object** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | To maintain robust and predictable code, it's best practice to assume that a moved-from object is only safe to be destroyed or assigned a new value, but not to be used in any meaningful way. |

| **Noncompliant Code** |
| --- |
| The integer values 0 through 9 are expected to be printed to the standard output stream from a std::string rvalue reference. However, because the object is moved and then reused under the assumption its internal state has been cleared, unexpected output may occur despite not triggering undefined behavior. |
| #include <iostream>  #include <string>    void g(std::string v) {  std::cout << v << std::endl;  }    void f() {  std::string s;  for (unsigned i = 0; i < 10; ++i) {  s.append(1, static\_cast<char>('0' + i));  g(std::move(s));  }  } |

| **Compliant Code** |
| --- |
| The std::string object is initialized to the expected value on each iteration of the loop. This practice ensures that the object is in a valid, specified state prior to attempting to access it in g(), resulting in the expected output. |
| #include <iostream>  #include <string>    void g(std::string v) {  std::cout << v << std::endl;  }    void f() {  for (unsigned i = 0; i < 10; ++i) {  std::string s(1, static\_cast<char>('0' + i));  g(std::move(s));  }  } |

| **Principle:**  Default Deny: By default, assume moved-from objects are not valid to prevent misuse. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 | DF4701, DF4702, DF4703 |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD  LANG.MEM.UVAR | Null Pointer Dereference  Uninitialized Variable |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-EXP63-a | Do not rely on the value of a moved-from object |
| Polyspace Bug Finder | R2024a | CERT C++: EXP63-CPP | Checks for read operations that reads the value of a moved-from object (rule fully covered) |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Constructing a std::string from a null pointer results in undefined behavior, which means the program might crash, produce incorrect results, or exhibit unpredictable behavior. To ensure safe and predictable behavior, always initialize a std::string with a valid C-string or use other safe construction methods. |

| **Noncompliant Code** |
| --- |
| A std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

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

| **Principle:**  Validate Input Data: This ensures that input data is validated to prevent null pointer dereferencing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2024.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Polyspace Bug Finder | R2024a | CERT C++: STR51-CPP | Check for string operations on null pointer (rule partially covered) |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent all SQL injection instances.** |
| --- | --- | --- |
| **SQL Injection** | [STD-004- CPP] | Preventing SQL injection is critical for securing applications that interact with databases. SQL injection occurs when an attacker can manipulate SQL queries by injecting malicious SQL code into an input field. Prepared statements ensure that SQL code is separated from data, which prevents attackers from injecting malicious SQL code. |

| **Noncompliant Code** |
| --- |
| This code modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement, but the prepared statement still permits SQL injection by incorporating a non-sanitized 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** |
| --- |
| The compliant solution uses a parametric query as a placeholder. The code also validates the length of the username, preventing an attacker from submitting a username that is too long. |
| 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:**  **Validate Input Data**: This ensures data is validated before being used in queries to avoid SQL injections.  **Default Deny**: This restricts what is allowed by default, ensuring that untrusted input is denied.  **Sanitize Data Sent to Other Systems**: This sanitizes data before it is used in external systems to prevent attacks like SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| 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 |
| Fortify | 1.0 | HTTP\_Response\_Splitting  SQL\_Injection\_\_Persistence  SQL\_Injection | Implemented |
| Klocwork | 2024.2 | SV.DATA.DB  SV.SQL  SV.SQL.DBSOURCE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and handle any memory allocation errors.** |
| --- | --- | --- |
| **Memory Protection** | [STD-005- CPP] | Memory allocation errors can occur when the system runs out of memory, and handling these errors gracefully can prevent a program from crashing unexpectedly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The is marked as noexcept, so the caller is going to assume that this function does not throw any exceptions, but because the operator can throw an exception if the allocation fails, it could lead to 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 will return either a null pointer or a pointer to allocated space. Always test the returned pointer to ensure it is not null before it is referenced. This solution handles this error condition appropriately when the returned pointer is null. |
| #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;  } |

| **Principles:**  **Practice Defense in Depth**: Adds an additional layer of security by ensuring memory issues are caught and managed.  **Use Effective Quality Assurance Techniques**: This ensures robustness by catching allocation errors during QA testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  |  |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how return values are handled |
| Helix QAC | 2024.2 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression.** |
| --- | --- | --- |
| **Assertions** | [STD-006- CPP] | Using static assertions in C++ allows for checking conditions at compile time, ensuring that certain invariants hold before the program even runs. This is particularly useful for validating the values of constant expressions. |

| **Noncompliant Code** |
| --- |
| This code uses the assert() macro to assert property concerning a memory mapped structure necessary 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** |
| --- |
| A preprocessor conditional statement may be used for assertions involving only constant expressions. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

| **Principles:**  **Heed Compiler Warnings**: Compiler warnings help catch errors early, ensuring code correctness.  **Use Effective Quality Assurance Techniques**: QA techniques like static assertions help verify assumptions in code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions.** |
| --- | --- | --- |
| **Exceptions** | [STD-007- CPP] | Handling exceptions properly is crucial for creating robust and reliable programs. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, neither f() or main() catch exceptions thrown, and 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 the 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  }  } |

| **Principles:**  **Adopt a Secure Coding Standard**: Standardizing error handling practices ensures that all potential issues are managed securely.  **Practice Defense in Depth**: Multiple layers of error handling prevent a single point of failure.  **Adhere to the Principle of Least Privilege**: Ensures that only the necessary amount of code is executed, and errors are handled without escalating privileges.  **Architect and Design for Security Policies**: Designing to catch and handle exceptions prevents security vulnerabilities from unhandled errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Klocwork | 2024.2 | MISRA.CATCH.ALL |  |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### 

#### Coding Standard 8

| **Coding Standard** | **Label** | **Use valid references, pointers, and iterators to reference elements of a basic\_string** |
| --- | --- | --- |
| Characters and Strings | [STD-008- CPP] | Always use references that are directly obtained from a valid std::string object, use pointers that are correctly derived from the address of a valid std::string element, and use iterators obtained from valid std::string member functions, and ensure they are not invalidated. |

| **Noncompliant Code** |
| --- |
| This 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 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 the compliant solution, the value of the iterator is updated as a result of each call to insert() so that the invalidated iterator is not accessed. The updated iterator is 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 : ' ');  }  } |

| **Principles:**  **Adopt a Secure Coding Standard**: Adhering to standards prevents undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.UAF** | Use After Free |
| Helix QAC | 2024.2 | **DF4746, DF4747, DF4748, DF4749** |  |
| Parasoft C/C++test | 2023.1 | **CERT\_CPP-STR52-a** | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2024a | CERT C++: STR52-CPP | Checks for use of invalid string iterator (rule partially covered). |

#### 

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not cast or delete pointers to incomplete classes** |
| --- | --- | --- |
| Expressions | [STD-009- CPP] | Casting pointers to incomplete classes can lead to undefined behavior because the compiler does not have enough information about the size and layout of the class. Without this information, it cannot correctly perform the cast. Deleting a pointer to an incomplete class is problematic because the destructor for the class will not be known to the compiler at the point of deletion. This can lead to incomplete destruction of the object, memory leaks, and undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, a class attempts to implement pimpl but deletes a pointer to an incomplete class type, resulting in undefined behavior if Body has a nontrivial destructor. |
| class Handle {  class Body \*impl; // Declaration of a pointer to an incomplete class  public:  ~Handle() { delete impl; } // Deletion of pointer to an incomplete class  // ...  }; |

| **Compliant Code** |
| --- |
| The deletion of impl is moved to a part of the code where Body is defined. |
| class Handle {  class Body \*impl; // Declaration of a pointer to an incomplete class  public:  ~Handle();  // ...  };    // Elsewhere  class Body { /\* ... \*/ };    Handle::~Handle() {  delete impl;  } |

| **Principles:**  **Keep It Simple**: Simplifies memory management, reducing complexity and potential errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **delete-with-incomplete-type** |  |
| Coverity | 6.5 | **DELETE\_VOID** | Fully implemented |
| Clang | 3.9 | -Wdelete-incomplete |  |
| CodeSonar | 8.1p0 | **LANG.CAST.PC.INC** | Conversion: pointer to incomplete |

#### 

#### Coding Standard 10

| **Coding Standard** | **Label** | **Overload allocation and deallocation functions as a pair in the same scope** |
| --- | --- | --- |
| Declarations and Initialization | [STD-010- CPP] | Overloading allocation and deallocation functions in pairs maintains consistency in how memory is allocated and deallocated. This ensures that the memory management strategy is coherent and that objects allocated with a custom new are deallocated with the matching custom delete. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example, an allocation function is overloaded at the global scope, but the corresponding deallocation function is not declared. If an object was to be allocated with the overloaded allocation function, any attempt to delete the object would result in undefined behavior. |
| #include <Windows.h>  #include <new>    void \*operator new(std::size\_t size) noexcept(false) {  static HANDLE h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }    // No corresponding global delete operator defined. |

| **Compliant Code** |
| --- |
| In the compliant solution, the corresponding deallocation function is also defined at global scope. |
| #include <Windows.h>  #include <new>    class HeapAllocator {  static HANDLE h;  static bool init;    public:  static void \*alloc(std::size\_t size) noexcept(false) {  if (!init) {  h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  init = true;  }    if (h) {  return ::HeapAlloc(h, 0, size);  }  throw std::bad\_alloc();  }    static void dealloc(void \*ptr) noexcept {  if (h) {  (void)::HeapFree(h, 0, ptr);  }  }  };    HANDLE HeapAllocator::h = nullptr;  bool HeapAllocator::init = false;    void \*operator new(std::size\_t size) noexcept(false) {  return HeapAllocator::alloc(size);  }    void operator delete(void \*ptr) noexcept {  return HeapAllocator::dealloc(ptr);  } |

| **Principles:**  **Architect and Design for Security Policies**: Ensures memory management is secure and consistent. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **new-delete-pairwise** | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-DCL54** |  |
| Clang | 3.9 | misc-new-delete-overloads | Checked with clang-tidy. |
| Helix QAC | 2024.2 | **C++2160** |  |

### 

### 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



During the Assess and Plan phase, it's essential to integrate security requirements early on. This means incorporating security principles such as "Validate Input Data" and "Architect and Design for Security Policies" right from the beginning to ensure security is embedded in the design. Automating the threat modeling process with tools like the Microsoft Threat Modeling Tool can help identify and mitigate potential threats early in the development lifecycle, aligning with secure coding standards.

In the Design phase, it's critical to adopt a security-driven approach, adhering to best practices like OWASP Top 10. To enforce standards such as "Default Deny" and "Principle of Least Privilege," components should be designed to reject unauthorized access by default and minimize roles and permissions. Additionally, automated design review tools can be implemented to ensure adherence to secure coding practices, such as "Sanitize Data Sent to Other Systems" and "Adopt a Secure Coding Standard," ensuring a secure design.

The Build phase is where security gets embedded into the codebase. Automating code analysis using tools like SonarQube or Checkmarx helps scan for compliance with standards like "Heed Compiler Warnings" and "Validate Input Data." These tools automatically flag violations of secure coding practices, ensuring that only secure code is integrated. Incorporating automated checks into the Continuous Integration (CI) pipeline further enforces these practices, ensuring that code passes essential security checks, such as input validation and exception handling, before it is merged.

Testing in the Verify and Test phase is crucial for identifying and fixing vulnerabilities before deployment. Integrating tools like OWASP ZAP for automated security testing helps enforce principles like "Practice Defense in Depth" by conducting vulnerability scans. This ensures any security gaps are identified and addressed. Automated tests that check for compliance with standards, such as handling memory allocation errors, further secure the codebase by ensuring robust and secure code is pushed to production.

As the software transitions to production in the Transition and Health Check phase, security must be verified through automated processes. Configuration management tools like Ansible or Puppet enforce security configurations and policies, aligning with "Architect and Design for Security Policies." Automated penetration tests should be regularly run to validate that systems are secure according to design specifications, ensuring a smooth and secure transition to production.

In the Monitor and Detect phase, ongoing monitoring is key to maintaining security in production. Automated Intrusion Detection Systems (IDS) and log monitoring tools can detect and alert security anomalies, supporting continuous security compliance. This automation helps enforce "Monitor and Detect" principles, allowing for real-time identification and response to potential security threats.

The Respond phase focuses on quickly addressing security incidents. Implementing automated response systems, including automated rollback procedures, helps in rapidly mitigating security breaches. This automated response enforces the "Default Deny" principle by quickly reverting systems to a secure state if a breach is detected, minimizing potential damage.

The Maintain and Stabilize phase ensures that security is continually upheld post-deployment. Continuous security baseline assessments using automated tools compare current configurations against established security baselines, enforcing ongoing security compliance. Regular updates and patches based on these assessments align with the "Maintain and Stabilize" principle, ensuring the system remains secure and stable over time.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | High | P3 | L3 |
| STD-002- CPP | Medium | Probable | Medium | P8 | L2 |
| STD-003- CPP | High | Likely | Medium | P18 | L1 |
| STD-004- CPP | High | Likely | Medium | P18 | L1 |
| STD-005- CPP | High | Likely | Medium | P18 | L1 |
| STD-006- CPP | Low | Unlikely | High | P1 | L3 |
| STD-007- CPP | Low | Probable | Medium | P4 | L3 |
| STD-008- CPP | High | Probable | High | P6 | L2 |
| STD-009- CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010- CPP | Low | Probable | Low | P6 | L2 |

### 

### 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 they are, 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 type of encryption ensures that data is secure while it is not being actively used, such as when it is stored on hard drives, databases, or cloud storage. Data at rest is typically encrypted using algorithms like AES (Advanced Encryption Standard) or RSA. This encryption can be applied to entire disks, specific files, or database entries. It involves converting the stored data into a cipher that can only be decrypted by authorized users with the correct decryption key. Encryption at rest is critical for protecting data from unauthorized access in the event of physical theft, unauthorized server access, or data breaches. |
| Encryption in flight | This type of encryption ensures that data is secure as it travels from one point to another. Data in flight is typically encrypted using protocols that encrypt data packets before they are transmitted over the network, making it difficult for unauthorized parties to intercept and read the information. Encryption in flight is crucial when transmitting sensitive or personal information over public or unsecured networks. |
| Encryption in use | This is the most complex form of encryption, as it involves securing data that is currently being computed on, without impacting the ability of systems to perform necessary operations. Encryption in use often involves the use of technologies that allow computations to be performed on encrypted data without decrypting it first. This ensures that data remains protected even while it is being processed in memory or used by applications. Encryption in use is important for scenarios where sensitive data needs to be processed, particularly in cloud computing environments or when working with highly confidential information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user, device, or system before granting access to resources. It ensures that the entity requesting access is who they claim to be. Authentication is implemented through mechanisms such as passwords, biometrics, two-factor authentication (2FA), or digital certificates. In a typical scenario, when a user attempts to log in to a system, they provide credentials like a username and password. The system checks the credentials against a stored database to confirm the user’s identity. More advanced methods like multi-factor authentication combine something the user knows with something they have, providing a higher level of security. Authentication is crucial in any environment where access to sensitive data or resources needs to be controlled. |
| Authorization | Authorization is the process of determining what an authenticated user, device, or system is allowed to do within the system. It ensures that users have appropriate permissions to access resources based on their identity. Once a user is authenticated, the system checks their permissions against an access control list or policy to determine what actions they can perform. Authorization mechanisms can range from role-based access control to more granular attribute-based access control. Authorization is critical for enforcing security policies that dictate who can access what within a system. It applies whenever there are different levels of access needed for different users or roles within an organization. |
| Accounting | Accounting, also known as auditing, is the process of recording and tracking user activities within a system. This includes logging access attempts, actions performed by users, and duration of sessions. Accounting is implemented through logging mechanisms that capture detailed records of user activities. These logs include information such as who accessed what, when, and from where, and what actions they performed. These logs are stored and can be analyzed for various purposes, including compliance reporting, detecting suspicious activities, or conducting forensic investigations after a security incident. Advanced accounting systems can also provide real-time monitoring and alerting when unusual activity is detected. Accounting is essential for maintaining an audit trail that can be used to ensure compliance with regulatory requirements, investigate security incidents, and provide evidence in case of disputes or legal actions. The policy applies in environments where monitoring and logging user activity are necessary for security, compliance, or operational reasons. |

## 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 | 08/05/2024 | Addition of standards and protocol | Victor Jarvis |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

**Sources/ Resources**

Carnegie Mellon University Software Engineering Institute’s SEI CERT C++ Coding Standard and assorted documentation provided a great deal of information used in this policy.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>