**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 | Ensure that all input data conforms to expected parameters in terms of length (to prevent buffer overflow errors) and format (to prevent SQL injection attacks) prior to passing it to any function. |
| 1. Heed Compiler Warnings | Select the highest warning level offered by the compiler and continue to edit and rewrite code until the compiler doesn’t issue any warnings. |
| 1. Architect and Design for Security Policies | Security should be an integral aspect of design and development of the system from the very beginning. The principles outlined in this document should act as a guideline for the *minimum* requirements, but the exact details of implementation of security practices will naturally vary from project to project. |
| 1. Keep It Simple | Aim for simplicity throughout the code to facilitate other developers’ fully grasping the purpose of the code and to minimize the potential for undue complexity to hide potential security issues: the simpler the code, the less scope there is for overlooked vulnerabilities to exist; and the more easily developers can understand the purpose of code, the more bandwidth they will have to spot any potential vulnerabilities therein. |
| 1. Default Deny | Access to any system, function or variable must be explicitly granted, otherwise it will be denied. Rigorous application of this principle allows much easier auditing of permissions (and the associated potential for security vulnerabilities) within the code base. |
| 1. Adhere to the Principle of Least Privilege | Ensure that access is granted to data or to code only where strictly necessary and that the permissions granted are as restrictive as possible while still allowing the programme to function. Again, rigorous application of this policy allows much easier auditing of privileges granted. |
| 1. Sanitize Data Sent to Other Systems | Examine data for special control characters or commands that could lead to a command being run on the other system: examples would include SQL injection attacks where a string that may appear innocuous in the context of the current system could be so tailored as to access data in a database that should not be accessible to that user. |
| 1. Practice Defense in Depth | Adopt a *Defence in Depth* strategy to ensure, wherever possible, that multiple security measures are taken so that if one should fail to prevent an attack others may nevertheless do so. |
| 1. Use Effective Quality Assurance Techniques | Ensure that unit tests are conducted against all new code and that integration testing is conducted upon introduction of new code to the codebase. Aim to automate testing where possible to improve efficiency and coverage and to reduce the possibility of human error. |
| 1. Adopt a Secure Coding Standard | Standards for secure coding should be agreed upon and all developers trained to employ the standards when creating code. This will ensure that everybody is on the same page as regards security measures and will keep the *Defence in Depth* strategy manageable across teams of developers. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Understand the data model used by your implementation(s). |

| **Noncompliant Code** |
| --- |
| This noncompliant example attempts to read a long into an int. This code works for models in which sizeof(int) == sizeof(long). For others, it causes an unexpected memory write similar to a buffer overflow. |
| int f(void) {  FILE \*fp;  int x;  /\* ... \*/  if (fscanf(fp, "%ld", &x) < 1) {  return -1; /\* Indicate failure \*/  }  /\* ... \*/  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the correct format for the type being used: |
| int f(void) {  FILE \*fp;  int x;  /\* Initialize fp \*/  if (fscanf(fp, "%d", &x) < 1) {  return -1; /\* Indicate failure \*/  }  /\* ... \*/  return 0;  } |

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

| **Principles(s):** This maps to the principle *Adopt a Secure Coding Standard*: ensuring that all developers understand the various datatypes — and that their precise implementation may vary — will prevent any unnecessary security vulnerabilities and defining proper use within a common standard can facilitate the creation of automated tests.  This coding standard also maps to the *Heed Compiler warnings* principle as most compilers would, at least if the strictest settings were invoked, identify such fast-and-loose use of data-types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT00 |  |
| PC-lint Plus | 1.4 | 559, 705, 706, 2403 | Assistance provided: Reports data type inconsistencies in format strings |
| Polyspace Bug Finder | R2023a | CERT C: Rec. INT00-C | Checks for:   * Use of basic numerical types instead of typedef-s * Integer overflow or integer constant overflow * Format string specifiers and arguments mismatch   Rec. partially covered. |
| PVS-Studio | 7.24 | V629, V5004 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not declare more than one variable per declaration. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a programmer or code reviewer might mistakenly believe that the two variables src and c are declared as char \*. In fact, src has a type of char \*, whereas c has a type of char. |
| char \*src = 0, c = 0; |

| **Compliant Code** |
| --- |
| In this compliant solution, each variable is declared on a separate line: |
| char \*src; /\* Source string \*/  char c; /\* Character being tested \*/ |

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

| **Principles(s):** This likewise maps to the principle *Adopt a Secure Coding Standard* as it would require ‘buy in’ from all developers in order to be implemented as desired. It also maps to *Keep it Simple* because the ease of reading and editing code is improved if this standard be observed, and that would make developers’ jobs easier when implementing other standards and adhering to other principles. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.DECL.ML | Multiple Declarations on Line |
| ECLAIR | 1.2 | CC2.DCL04 | Fully implemented |
| LDRA tool suite | 9.7.1 | 579 S | Fully implemented |
| Parasoft C/C++test | 2022.2 | CERT\_C-DCL04-a | Each variable should be declared in a separate declaration statement |

#### Coding Standard 3

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

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

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

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

| **Principles(s):** This standard would map to the principles *Validate User Input* and *Sanitize Data Sent to Other Systems* because these are two major vectors of attack used by would-be hackers: an attack could be launched by causing a buffer overflow when inputting data, or the data input (even if it does not result in an overflow) could cause malicious code to run if passed to another system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow  Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| CodeSonar | 7.3p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |

#### Coding Standard 4

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

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a char array, the database connection is created, and then the passwords are hashed.  Unfortunately, this code example permits a SQL injection attack by incorporating the unsanitized input argument username into the SQL command, allowing an attacker to inject validuser' OR '1'='1. The password argument cannot be used to attack this program because it is passed to the hashPassword() function, which also sanitizes the 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 + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString);    if (!rs.next()) {  throw new SecurityException(  "User name or password incorrect"  );  }    // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  }  }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long user name. |
| public void doPrivilegedAction(  String username, char[] password  ) throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  try {  String pwd = hashPassword(password);    // Validate username length  if (username.length() > 8) {  // Handle error  }    String sqlString =  "select \* from db\_user where username=? and password=?";  PreparedStatement stmt = connection.prepareStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, pwd);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("User name or password incorrect");  }    // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  }  } |

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

| **Principles(s):** This standard maps to the principles *Architect and Design for Security Policies* because it recognizes the importance of security being ‘baked-in’ to the very structure of the code itself rather than being superimposed upon it as an afterthought. It also maps, again, to the principle of *Sanitize Data sent to other Systems* because it acknowledges that data that could appear innocuous in one system could be very damaging when sent to another. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Fortify | 1.0 | HTTP\_Response\_Splitting  SQL\_Injection\_\_Persistence  SQL\_Injection | Implemented |
| Klocwork |  | SV.DATA.BOUND  SV.DATA.DB  SV.HTTP\_SPLIT  SV.PATH  SV.PATH.INJ  SV.SQL | Implemented |
| SonarQube | 6.7 | S2077  S3649 | Executing SQL queries is security-sensitive  SQL queries should not be vulnerable to injection attacks |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Explicitly construct and destruct objects when manually managing object lifetime |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a class with nontrivial initialization (due to the presence of a user-provided constructor) is created with a call to std::malloc(). However, the constructor for the object is never called, resulting in undefined behavior when the class is later accessed by calling s->f(). |
| #include <cstdlib>    struct S {  S();    void f();  };    void g() {  S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));    s->f();    std::free(s);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructor and destructor are both explicitly called. Further, to reduce the possibility of the object being used outside of its lifetime, the underlying storage is a separate variable from the live object. |
| #include <cstdlib>  #include <new>    struct S {  S();    void f();  };    void g() {  void \*ptr = std::malloc(sizeof(S));  S \*s = new (ptr) S;    s->f();    s->~S();  std::free(ptr);  } |

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

| **Principles(s):** This maps to *Adopt a Secure Coding Standard* because it details the specifics of how a certain element of code (constructors and destructors, in this case) should be handled by all developers to ensure that the code is as secure as possible. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.1 | DF4761, DF4762, DF4766, DF4767 |  |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-MEM53-a | Do not invoke malloc/realloc for objects having constructors |
| Polyspace Bug Finder | R2023a | CERT C++: MEM53-CPP | Checks for objects allocated but not initialized (rule fully covered). |
| PVS-Studio | 7.24 | V630, V749 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Understand the termination behavior of assert() and abort(). |

| **Noncompliant Code** |
| --- |
| This noncompliant code example defines a function that is called before the program exits to clean up: |
| void cleanup(void) {  /\* Delete temporary files, restore consistent state, etc. \*/  }    int main(void) {  if (atexit(cleanup) != 0) {  /\* Handle error \*/  }    /\* ... \*/    assert(/\* Something bad didn't happen \*/);    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to assert() is replaced with an if statement that calls exit() to ensure that the proper termination routines are run: |
| void cleanup(void) {  /\* Delete temporary files, restore consistent state, etc. \*/  }    int main(void) {  if (atexit(cleanup) != 0) {  /\* Handle error \*/  }    /\* ... \*/    if (/\* Something bad happened \*/) {  exit(EXIT\_FAILURE);  }    /\* ... \*/  } |

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

| **Principles(s):** This standard maps to the *Architect and Design for Security Policies* principle because it deals with specifics of code architecture as it relates to security implications and provides specific guidance to developers on how to use this macro in C++. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/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 | 9.7.1 | 44 S | Enhanced enforcement |
| Parasoft C/C++test | 2022.2 | CERT\_C-ERR06-a | Do not use assertions |
| PC-lint Plus | 1.4 | 586 | Fully supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception objects must be nothrow copy constructible |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an exception of type S is thrown in f(). However, because S has a std::string data member, and the copy constructor for std::string is not declared noexcept, the implicitly-defined copy constructor for S is also not declared to be noexcept. In low-memory situations, the copy constructor for std::string may be unable to allocate sufficient memory to complete the copy operation, resulting in a std::bad\_alloc exception being thrown. |
| #include <exception>  #include <string>    class S : public std::exception {  std::string m;  public:  S(const char \*msg) : m(msg) {}    const char \*what() const noexcept override {  return m.c\_str();  }  };    void g() {  // If some condition doesn't hold...  throw S("Condition did not hold");  }    void f() {  try {  g();  } catch (S &s) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the type of the exception object can inherit from std::runtime\_error, or that type can be used directly. Unlike std::string, a std::runtime\_error object is required to correctly handle an arbitrary-length error message that is exception safe and guarantees the copy constructor will not throw [ ISO/IEC 14882-2014 ]. |
| #include <stdexcept>  #include <type\_traits>    struct S : std::runtime\_error {  S(const char \*msg) : std::runtime\_error(msg) {}  };    static\_assert(std::is\_nothrow\_copy\_constructible<S>::value,  "S must be nothrow copy constructible");    void g() {  // If some condition doesn't hold...  throw S("Condition did not hold");  }    void f() {  try {  g();  } catch (S &s) {  // Handle error  }  } |

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

| **Principles(s):** This maps to the principle *Adopt a Secure Coding Standard* because it outlines the proper construction of exception objects which all developers should follow to ensure that error hiding — which could obscure other potential security vulnerabilities— is avoided. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | cert-err60-cpp | Checked by clang-tidy |
| Helix QAC | 2023.1 | C++3508 |  |
| Parasoft C/C++test | 2022.2 | CERT\_CPP-ERR60-a  CERT\_CPP-ERR60-b | Exception objects must be nothrow copy constructible  An explicitly declared copy constructor for a class that inherits from 'std::exception' should have a non-throwing exception specification |
| PRQA QA-C++ | 4.4 | 3508 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Lifetime | [STD-008-CPP] | Do not access an object outside of its lifetime. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in undefined behavior. |
| struct S {  void mem\_fn();  };    void f() {  S \*s;  s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, storage is obtained for the pointer prior to calling S::mem\_fn(). |
| struct S {  void mem\_fn();  };    void f() {  S \*s = new S;  s->mem\_fn();  delete s;  } |

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

| **Principles(s):** This standard maps to the principle *Use Effective Quality Assurance Techniques* because thorough unit testing should expose any errors resulting from the undefined behaviour caused by violating this standard. It also maps to *Architect and Design for Security Policies* because it specifies how object lifetimes should be handled in order to avoid potential security issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | return-reference-local  dangling\_pointer\_use | Partially checked |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<> |
| CodeSonar | 7.3p0 | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| Helix QAC | 2023.1 | C++4003, C++4026  DF2812, DF2813, DF2814, DF2930, DF2931, DF2932, DF2933, DF2934, |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Guarantee that container indices and iterators are within the valid range. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #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):** This maps to the principle *Validate User Input* as it deals with the potential for malicious users to manipulate the code by inputting unexpected data. It also maps to *Architect and Design for Security Policies* because it is concerned with the architecture of the code itself and how it might be attacked by would-be hackers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| Helix QAC | 2023.1 | C++3139, C++3140  DF2891 |  |
| LDRA tool suite | 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 | 2002.2 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | [STD-010-CPP] | Do not access a closed file. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the stdout stream is used after it is closed: |
| #include <stdio.h>    int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }    printf("stdout successfully closed.\n");  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, stdout is not used again after it is closed. This must remain true for the remainder of the program, or stdout must be assigned the address of an open file object. |
| #include <stdio.h>    int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }    fputs("stdout successfully closed.", stderr);  return 0;  } |

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

| **Principles(s):** This standard maps to the principles *Use Effective Quality Assurance Techniques* and *Adopt a Secure Coding Standard* because this is a standard which all developers should be following throughout their work, and is a standard which comprehensive unit testing could help to enforce by flagging up any oversights |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported |
| CodeSonar | 7.3p0 | IO.UAC | Use after close |
| Compass/ROSE |  |  |  |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Implemented |

### 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.

Automation can be implemented at the *Build* stage by verifying that all libraries and repositories are still trusted and actively maintained to avoid exposing the system to risk through deficient library code.

The *Verify and Test* stage can also benefit from substantial automation by utilizing unit tests to verify code integrity and ensuring that code conforms to the standards laid out in this document.

Penetration testing — as part of the *Transition and Health Check* stage — can also be at least partially automated to determine the system’s robustness in the face of a simulated attack. These latter two stages would likely identify many failures in meeting the standards proposed herein, whilst also highlighting opportunities for improvement in the face of developing threats not previously considered.

Finally, automated log collection, analysis and intrusion detection at the *Monitor and Detect* stage could facilitate rapid releases of security patches and updates following identification of security issues that might arise once the system is in production.

Implementing security procedures at multiple stages in this way provides *Defense in Depth* by creating multiple opportunities to identify any given issue so that if it should be missed at one stage it may yet be caught at the next.

### 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 | Unlikely | High | P3 | L3 |
| STD-002-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Probable | Medium | P12 | 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 | High | Probable | High | P6 | L2 |
| STD-009-CPP | High | Likely | High | P9 | L2 |
| STD-010-CPP | Medium | Unlikely | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest refers to securing data that is being stored even though it may not be being accessed or transmitted. Both servers and individuals’ laptops and desktop machines will utilize full-disk encryption in all cases. Database encryption will also be applied across the board. These measures will prevent theft of data or unauthorized access to systems in the event that a developer’s computer be stolen or the database be hacked. It would also be prudent to limit storage of the most sensitive data to as few locations as possible to *Keep it Simple* when it comes to securing that data. |
| Encryption at flight | Encryption in Flight describes protecting data which is to be transmitted over a network. All data being transmitted will be encrypted using public-key cryptography to prevent anybody attempting to intercept the data being able to decipher it. Use secure channels such as *scp* or *sftp* for file transfer, rather than *ftp* which is not secure. |
| Encryption in use | Encryption in use refers to protection of data as it is being accessed and used. The same principles outlined for *Encryption in Flight* would apply here. Access to data should also be controlled, as laid out in the AAA policy, to ensure that only users that really need the data are actually able to access it. AI security analytics tools can monitor aggregate data about access to data to quickly identify any anomalous activity, freeing up admin staff to look into those possible breaches rather than having to identify them in the first place. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication describes the use of user logins and other mechanisms to prevent access to the project and its associated systems by people not granted access. This would be the first line of defense in a *Defense in Depth* strategy as outlined in the above principles. Allowing access only to those who are involved in the project reduces (at least in principle: authentication is only one layer of the DiD) the number of people who need be taken into account when limiting access to sensitive aspects of the project, as outlined below in *Authorization*. |
| Authorization | Authorization limits authenticated users’ access only to those systems which they have been approved as having a legitimate reason to access and permits only those actions which it is deemed appropriate for said user to perform, in line with the principles *Default Deny* and *Adhere to the Principle of Least Privilege.* This is important to ensure that access to key systems and the ability to perform certain actions can be tailored to each individual user depending upon their area of expertise, the extent of their professional experience and the extent to which they are trusted, rather than relying on a one-size-fits all authenticated-or-not binary which would not admit of any granularity and hence would effectively open the entire system to anybody involved at any level. |
| Accounting | Accounting describes the process of monitoring what action is being taken by whom and what data is transferred in and out of the system. This layer of defense complements both *Authentication* and *Authorization* because it would allow administrators to verify that those two processes are working as they should. However, it goes deeper than that by also providing a *reactive* mechanism (whereas the other elements of AAA are *proactive*) which allows breaches to be detected and dealt with much more quickly: the previous two steps mean that access to the system by any unauthenticated party or actions taken by a user not authorized to take them would be relatively easy to detect; and this allows administrators to free up more resources to audit the actual use of systems by those users authorized to use them to make sure that they are not misusing the trust placed in them. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 03/18/2023 | Completed details of security principles and proposed ten coding standards | Ed Morrow | Prof. Prasad |
| 1.9 | 04/09/2023 | Completed risk assessment, automation tools and encryption and AAA policies | Ed Morrow | Prof. Prasad (pending) |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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