**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 | Test inputs ensure that only legitimate data, and not erroneous or malicious data, enters an information system. |
| 1. Heed Compiler Warnings | These alerts are there to inform a developer of a potential bug or issue in the code. While warnings do not prohibit code from compiling, errors do, therefore both are equally vital to take into account when changing code to reduce potential security issues. |
| 1. Architect and Design for Security Policies | When designing security policies, take into account the software architecture and design, such as dividing a system into subsystems with various degrees of authorization or privilege. |
| 1. Keep It Simple | Errors in coding and use are less likely to occur when designs are kept straightforward and minimal. This idea may also reduce the amount of security complexity needed. |
| 1. Default Deny | By default, access is prohibited, and access is only made possible under the terms of the protection system being utilized. |
| 1. Adhere to the Principle of Least Privilege | Processes should run with the fewest rights necessary to finish the work, and higher privileges should be used as little as possible and quickly as possible. This lessens the possibility of an attacker executing code with high privileges. |
| 1. Sanitize Data Sent to Other Systems | For example, SQL injection attacks, unused functions, or calls made out of context, may pass and harm a system. Prior to activating other systems, these possible problems are checked by data sanitization. |
| 1. Practice Defense in Depth | Have several levels of defense so that, should one layer of defense become vulnerable, potential exploits or damage can be mitigated. |
| 1. Use Effective Quality Assurance Techniques | An efficient Quality Assurance program can include proper testing, such as fuzz and penetration testing, as well as audits of the code. Internal and external security reviews can aid in identifying and resolving potential problems. |
| 1. Adopt a Secure Coding Standard | To be secure from the start, follow coding standards in the language and platform of your choosing. |

### 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** | NUM00-J | Programs must not allow mathematical operations to exceed the integer ranges provided by their primitive integer data types. |

| **Noncompliant Code** |
| --- |
| The operation of the code below will cause an overflow, which will produce inaccurate results and possibly expose the software to security flaws. |
| public static int multAccum(int oldAcc, int newVal, int scale) {  // May result in overflow  return oldAcc + (newVal \* scale);  } |

| **Compliant Code** |
| --- |
| When doing mathematical operations, the use of procedures like safeAdd() or safeMultiply() ensures proper exception throwing. |
| public static int multAccum(int oldAcc, int newVal, int scale) {  return safeAdd(oldAcc, safeMultiply(newVal, scale));  } |

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

| **Principles(s):** Maintain Simplicity. It is best to use integers and buffers in a straightforward manner without performing intricate calculations that could lead to mistakes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | BAD\_SHIFT\_OVERFLOW\_BEFORE\_WIDEN | Implemented |
| Parasoft Jtest | 2020.2 | PB.NUM.ICO  PB.NUM.BSA  PB.NUM.CACO | Stay away from calculations that produce overflow or NaN  Use [0, 31] as the range for all integers when determining the magnitude of a shift.  Whenever possible, avoid employing compound assignment operators because they can lead to overflow |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | CTR51-CPP | Iterators, a generalization of pointers, enable a C++ program to interact uniformly with various data structures (containers) [ISO/IEC 14882-2014]. It is necessary to reference values through a valid iterator, pointer, or reference, which is a close relationship between pointers, references, and iterators. |

| **Noncompliant Code** |
| --- |
| Code that changes semicolons in a common string object to spaces. noncompliant due to the first call to the insert() function. |
| #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** |
| --- |
| Every time insert() is used, the value of the iterator, loc, is updated, ensuring that an invalid iterator is never accessed. |
| #include <string>  void f(const std::string &input) {  std::string email  // Copy input into email converting “;” to “ “  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(), i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ‘;’ ? \*i : ‘ ‘);  }  } |

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

| **Principles(s):** Design and Architect for Security Policies. It's crucial to write your code in a way that prevents frequent mistakes, such the memory management problems pointers in C++. Use pointers appropriately for common data types like basic\_strings while designing programs to be as secure as possible. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-STR52-a | To refer to items of basic\_string, use only proper references, pointers, and iterators. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | Assure that character data and the null terminator have enough space in string storage. Buffer overflow can result in major security flaws, particularly when utilizing C and C++-based languages. It's crucial to ensure proper processing of strings with enough memory in addition to validating input. |

| **Noncompliant Code** |
| --- |
| Because the 'cin' function reads unbounded input, it is not advised to read input using just this function. |
| #include <iostream>  void f() {  char buf[12];  std::cin > buf;  } |

| **Compliant Code** |
| --- |
| The easiest technique to prevent buffer overflow is to use std::string rather than a buffer array. String objects can be read into and then modified as necessary. |
| #include <iostream>  #include <string>  void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Check the input data. The system often accepts user input in the form of strings (consider usernames and passwords). It is crucial to check the buffer size as well as the proper handling and storing of string variables throughout the entire system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Klocwork | 2018 | NNTS.MIGHT  NNTS>TAINTED |  |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Polyspace Bug Finder | R2020a | CERT C++: STR50-CPP | Using a risky standard function  Incorrect string format specifier causes a buffer overflow and a missing null in a string array.  string manipulation resulting in a destination buffer overflow  Rule only partially applied |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS00-J | Prevent SQL Injection by first sanitizing and validating user input and then double-checking input from other sources (such as other APIs, subsystems, etc.). |

| **Noncompliant Code** |
| --- |
| The following code exposes the application to SQL Injection by using the keyword "username" as an uncleansed parsing for user 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 = PropertyManaer.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 doPrivilgedAction(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.createSatement();  ResusltSet rs = stmt.executeQuery(sqlString);  if (!rs.next()) {  throw new SecurityException(“User name or password incorrect”);  }  // Authenticated, proceed  } finally {  try {  connection.close();  } catch(SQLExeption x) {  // Forward to handler  }  }  }  } |

| **Compliant Code** |
| --- |
| SQL Injection is avoided by using parametric queries with a? character as a placeholder. |
| 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):** Check the input data. The majority of SQL Injection attacks take place in portions of the program where users must log in or verify their identities. Verifying the kind of data being input into the system can help reduce the likelihood of these attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| Findbugs | 1.0 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| SonarQube | 6.7 | S2077  S3649 | Security-sensitive SQL query execution shouldn't leave SQL queries open to injection attacks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM50-CPP | Never use freed memory. A major contributor to memory leak in applications is pointer arithmetic, particularly when it comes to the use and release of memory within a program. Such dangling pointers can create security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| After being deallocated, the pointer is dereferenced, leaving the code vulnerable to attacks. |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // …  delete s;  // …  s->f();  } |

| **Compliant Code** |
| --- |
| Memory that is dynamically allocated is not deallocated until it is no longer needed. |
| #include <new>  struct S {  void f();  }  void g() noexecpt(false) {  S \*s = new S;  // …  s->f();  delete s;  } |

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

| **Principles(s):** Maintain Simplicity. Buffer overflows and other mistakes, particularly in complex systems, can result from poor memory management. These mistakes could expose the system to risks. To ensure the appropriate maintenance of the application, effectively and simply manage memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | dangling\_pointer\_use |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | can identify particular instances where memory is read from or written to the target of a freed pointer or deallocated more than once. |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL03-C | To check the value of a constant expression, use a static assertion. Even while it is useful, the simple assert() macro has some drawbacks, particularly for runtime and server-side programs. At compile time, error messages are provided as string objects by the static\_assert() function. |

| **Noncompliant Code** |
| --- |
| To verify a memory-mapped structure's property, use the assert() macro. |
| #include <assert.h>  struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  }  in func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| Preprocessor conditional statement used to assert constant expression. |
| 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 |

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

| **Principles(s):** Utilize efficient quality assurance methods. This includes using testing software and code correctly, such as the assert() macro. To ensure the long-term functionality of your application, use the best tools possible when performing quality assurance testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Utilizing clang-tidy |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| CodeSonar | 6.0p0 | (customization) | The assert() macro can be used in checks that the user can implement. |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR50-CPP | Don't end the program suddenly. By calling exit handlers or object destructors, functions like std::abort() and std::\_Exit() shut down programs without using exit handlers. Due to the potential risk involved, they should only be called in the event of a serious application fault. |

| **Noncompliant Code** |
| --- |
| Although the f() function is used as an exit handler, it has the potential to fail and cause exceptions to be issued. |
| #include <cstdlib>  void throwing\_func() noexecpt(false);  void f() { // Not invoked by program except as exit handler  throwing\_func();  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // …  } |

| **Compliant Code** |
| --- |
| f() handles all exceptions issued by the throwing\_func() |
| #include <cstdlib>  void throwing\_func() noexecpt(false);  void f() {  try {  throwing\_func();  } catch(...) {  // Handle error  }  } |

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

| **Principles(s):** Architect and Design for Security Policies. Whether you restrict functionality completely or simply throw the user a specific error, exception management is a crucial component of effective software design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | stdlib-use | Partially checked |
| CodeSonar | 6.0p0 | BADFUNC.ABORT  BADFUNC.EXIT | Using abort and exit |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced Enforcement |
| RuleChecker | 20.10 | stdlib-use | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| I/O | FIO51-CPP | When a file is not needed anymore, close it. To stop memory leaks in the application, pointers and the objects (such files) they point to must be managed properly. If the software no longer requires access to a file, make sure to close it appropriately. |

| **Noncompliant Code** |
| --- |
| Although the program is created and read into memory, constructors are not called since the program is terminated. Consequently, the file won't be closed. |
| #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** |
| --- |
| std::fstream::close() is called before terminating program. |
| #include <exception>  #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ..  file.close();  if (file.fail()) { |

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

| **Principles(s):** Keep it simple. Multiple file usage, opening, and manipulation systems might get complex. All areas of the application should adhere to the security practice of treating files and the pointers that point to them in memory properly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2020a | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| OOP | OOP50-CPP | Do not use constructors or destroyers to call virtual functions. Calling a derived class function from a base function call is risky (i.e., resource management difficulties), as the order of development starts with base classes and continues to derived classes. |

| **Noncompliant Code** |
| --- |
| Do not use constructors or destroyers to call virtual functions. Calling a derived class function from a base function call is risky (i.e., resource management difficulties), as the order of development starts with base classes and continues to derived classes. |
| struct B {  B() { seize(); }  virtual ~B() { release(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  B::seize();  // Get derived resources…  }  void release() override {  // Release derived resources…  B::release();  }  }; |

| **Compliant Code** |
| --- |
| calls made to private and nonvirtual member functions rather than virtual functions. |
| class B {  void seize\_mine();  void release\_mine();  public:  B() { seize\_mine(): }  virtual ~B() { release\_mine(); }  protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };  class D : public B {  void seize\_mine();  void release\_mine();  public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }  protected:  void seize() override {  B::seize();  seize\_mine();  }  void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):** Keep it simple, any application that applies OOP principles inevitably becomes more complex. To maintain code clean and easier to manage, it is important to follow the best, most accepted OOP principles, such as knowing when to use (and when not to use) virtual function calls. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | virtual-call-in-constructor | Fully checked |
| Clang | 3.9 | clang-analyzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| PVS-Studio | 7.07 | V1053 |  |
| RuleChecker | 20.10 | virtual-call-in-constructor | Fully checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Random Numbers | MSC50-CPP | Never produce pseudorandom numbers with std::rand(). The quality of the generated random number sequence cannot be guaranteed by using the standard library function from the C library. Certain sequences may be brief and simple to predict. |

| **Noncompliant Code** |
| --- |
| Use the rand() function to create an ID. Because ID is predictable and has little randomization, it exposes applications to security flaws. |
| #include <cstdlib>  #include <string>  void f() {  std::string id(“ID”); // Holds ID starting with characters “ID” followed  // by random int in the range [0-100]  id += std::to\_string(std::rand() % 10000);  // …  } |

| **Compliant Code** |
| --- |
| By separating the process into two steps—an algorithm producing random values and dispersing those values using a density function—the C++ library <random> allows for more control over random number production. |
| #include <random>  #include <string>  void f() {  std::string id(“ID”);  std::uniform\_int\_distribution<int> distribution(0, 10000);  std::random\_device rd;  std::mt19937 engine(rd());  id += std::to\_string(distribution(engine));  // …  } |

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

| **Principles(s):** Architect and Design for Security Policies. The most recent coding standards should be used by developers, such as using library functions in C++ instead of some dated, less secure routines from the standard library. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | bad-function (AUTOSAR.26.5.1A) | Fully checked |
| Clang | 4.0 (prerelease) | cert-msc50-cpp | Checked by clang-tidy |
| ECLAIR | 1.2 | CC2.MSC30 | Fully implemented |
| Polyspace Bug Finder | R2020a | CERT C++: MSC50-CPP | Verifies whether a weak pseudo-random number generator is being used (partially covered rule). |

### 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.

[Insert your written explanations here.]

### 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 |
| --- | --- | --- | --- | --- | --- |
| NUM00-J | Medium | Unlikely | Medium | 4 | 3 |
| CTR51-CPP | High | Probable | High | 6 | 2 |
| STR50-CPP | High | Likely | Medium | 18 | 1 |
| IDS00-J | High | Probable | Medium | 12 | 1 |
| MEM50-CPP | High | Likely | Medium | 18 | 1 |
| DCL03-C | Low | Unlikely | High | 1 | 3 |
| ERR50-CPP | Low | Probable | Medium | 4 | 3 |
| FIO51-CPP | Medium | Unlikely | Medium | 4 | 3 |
| OOP50-CPP | Low | Unlikely | Medium | 2 | 3 |
| MSC50-CPP | Medium | Unlikely | Low | 6 | 2 |

### 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 | relates to the encryption of non-currently used data. For instance, it's crucial to make sure that data is encrypted and secure if it contains a database of usernames, passwords, or other potentially sensitive information that isn't constantly used by your system. Far too frequently, data breaches that make headlines are caused by data being stored in plain text or without any kind of security. |
| Encryption at flight | is the action of encrypting data entering or leaving your system. Examples include sending emails and using the public internet. It is crucial that the data in the system is encrypted at all times, including when it enters and exits the system from outside sources as well as when it is transferred between system components and modules. |
| Encryption in use | the security and encryption of the data that the user is currently accessing. This entails creating, viewing, and editing data in a safe, encrypted environment. The most common instance of data in use being encrypted would be during password verification. An example of encryption in use is the usage of a hashing function to encrypt the password and a decryption function to read, translate, and validate the data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of authenticating a user who is attempting to access a system involves confirming their identity. Anyone accessing the system, whether they are customers or workers, should have a username and password combination that can be confirmed by the system (ideally through an encrypted database of username/password combinations). Every user of the system should ideally have some type of secondary authentication, such as trusted devices or unique codes delivered to confirmed points of contact. |
| Authorization | After a user has successfully authenticated into the system, authorization is the process of verifying and authorizing them access. Administrative users should, for instance, be able to alter the database, conduct queries, and execute system-level commands as necessary. In contrast to the general consumer/user-level, who should only have access to a limited number of files and programs (with minimum of permissions to see and operate things), they should also be able to add people to the system. |
| Accounting | The process of keeping track of and being accountable for the system's and its data's digital well-being is known as accounting, also known as auditing. The simplest illustration of this is a log file, which keeps track of system activity such as data accesses and edits along with the user account responsible for such changes. This is an important aspect of digital security because it gives engineers a clear picture of who the attacker was, how they gained access to the system, and a plan for how to effectively defend against repeat attacks in the event that the system is subject to unauthorized access or attack. |

**\***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 | 07/13/2023 | Milestone - Coding Standards | Jorge Argueta |  |
| 1.2 | 08/05/2023 | Project One: Security Policy | Jorge Argueta |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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