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



Green Pace Secure Development Policy

Contents

[Overview 2](#_30j0zll)

[Purpose 2](#_1fob9te)

[Scope 2](#_3znysh7)

[Module Three Milestone 2](#_2et92p0)

[Ten Core Security Principles 2](#_tyjcwt)

[C/C++ Ten Coding Standards 3](#_1t3h5sf)

[Coding Standard 1 4](#_4d34og8)

[Coding Standard 2 5](#_2s8eyo1)

[Coding Standard 3 6](#_17dp8vu)

[Coding Standard 4 7](#_3rdcrjn)

[Coding Standard 5 8](#_26in1rg)

[Coding Standard 6 9](#_lnxbz9)

[Coding Standard 7 10](#_35nkun2)

[Coding Standard 8 11](#_1ksv4uv)

[Coding Standard 9 13](#_44sinio)

[Coding Standard 10 14](#_2jxsxqh)

[Defense-in-Depth Illustration 15](#_z337ya)

[Project One 15](#_3j2qqm3)

[1.](#_1y810tw) Revise the C/C++ Standards 15

[2.](#_4i7ojhp) Risk Assessment 15

[3.](#_2xcytpi) Automated Detection 15

[4.](#_1ci93xb) Automation 15

[5.](#_3whwml4) Summary of Risk Assessments 16

[6.](#_2bn6wsx) Create Policies for Encryption and Triple A 16

[7.](#_qsh70q) Map the Principles 17

[Audit Controls and Management 18](#_3as4poj)

[Enforcement 18](#_1pxezwc)

[Exceptions Process 18](#_49x2ik5)

[Distribution 19](#_2p2csry)

[Policy Change Control 19](#_147n2zr)

[Policy Version History 19](#_3o7alnk)

[Appendix A Lookups 19](#_23ckvvd)

[Approved C/C++ Language Acronyms 19](#_ihv636)

# 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 | This principle focuses on validating and checking ALL input to the system, especially those input by the user or other external sources. Most issues with modern software come from non-validated input. |
| 1. Heed Compiler Warnings | This principle focuses on the use of static and dynamic tools to compile code and check for warnings. It is best practice to use the highest warning levels possible and to modify the code bit by bit to eliminate all warnings. |
| 1. Architect and Design for Security Policies | This principle focuses on the design of the software, specifically with the intent of implementing and enforcing security policies. The developer(s) should modularize the software as needed to ensure all aspects are properly enforced. |
| 1. Keep It Simple | This principle is self-explanatory: keep the software design and implementation as simple as possible. More complicated programs tend to have more, and harder to find, errors. Additionally, simple designs help with future maintenance of the system by multiple developers. |
| 1. Default Deny | Another self-explanatory principle: deny all access to system resources, function calls, etc. and only provide access on a case-by-case basis. All interactions should be denied by default, and only allow access under certain conditions (such as privileged user logins, specific permissions on parts of the system, etc.). |
| 1. Adhere to the Principle of Least Privilege | Similar in fashion to the previous principle. Every process of the software should execute with the lowest level of privilege as possible. If, for whatever reason, elevated privileges are required, they should be used for the least amount of time required. |
| 1. Sanitize Data Sent to Other Systems | This principle is similar, but not quite the same, as “Input Validation”. In more complex systems, process calling isn’t aware of the context in which data is being passed in: therefore, any subsystem that sends or receives information from other sources should sanitize, or validate, information being passed throughout the system. |
| 1. Practice Defense in Depth | Defense in Depth refers to the layering of multiple different defense strategies to provide the highest level of security to the system as possible. It is good practice to use and implement various layers of defense (such a secure programming, firewalls and network security, or securing the runtime environment) so that, should one area fall victim to a vulnerability, the others will protect the rest of the system. |
| 1. Use Effective Quality Assurance Techniques | This principle focuses on testing the system/software in various ways (such as Fuzz testing, penetration testing, etc.) to further eliminate vulnerabilities and any unseen or unexpected use-cases. This can be done both internally by the developers or externally by independent security reviews. |
| 1. Adopt a Secure Coding Standard | This principle is the umbrella that catches the following section of the security policy: developers should adopt and implement coding standards based on the language and platform being used. |

## 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 | Detect or prevent integer overflow. Programs involving mathematical operations, especially with primitive data types, should take steps to ensure said operations do not exceed the range from variable types. |

|  |
| --- |
| **Noncompliant Code** |
| Operation of the below code will result in an overflow, which 1) will provide incorrect results and 2) potentially open up the program to security vulnerabilities. |
| public static int multAccum(int oldAcc, int newVal, int scale) {  // May result in overflow  return oldAcc + (newVal \* scale);  } |

|  |
| --- |
| **Compliant Code** |
| Use of functions such as safeAdd() or safeMultiply() ensure proper exception throwing when using mathematical operations. |
| 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):** Keep It Simple. Integers and buffers should be used in a clear, simple way without complicated, and potentially error-inducing calculations. |

**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 | Avoid calculations which result in overflow or NaN  Do not use an integer outside the range of [0, 31] as the amount of a shift  Avoid using compound assignment operators in cases whic h may cause overflow |

### Coding Standard 2

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Data Value** | CTR51-CPP | Use valid references, pointers, and iterators to reference elements of a basic\_string. Pointer error is a common source of security vulnerabilities in C++, so ensuring pointers point to valid data values is important. |

|  |
| --- |
| **Noncompliant Code** |
| Code to replace semicolons with spaces in a standard string object. Noncompliant due to first call to 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** |
| Value of the iterator, loc, is updated each time insert() is called, therefore never accessing an invalid iterator. |
| #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):** Architect and Design for Security Policies. It is important to design your code in a way that avoids common errors, like memory management issues with pointers in C++. Design code in the most secure way possible, which includes proper use of pointers for common data types like basic\_strings. |

**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 | Use valid references, pointers, and iterators to reference elements of basic\_string |

### Coding Standard 3

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **String Correctness** | STR50-CPP | Guarantee that storage for strings ahs sufficient space for character data and the null terminator. Buffer overflow can cause serious vulnerabilities, especially when using languages such as C and C++. It is important not only to validate input, but to ensure proper handling of strings with sufficient memory storage. |

|  |
| --- |
| **Noncompliant Code** |
| Reading input using a standard ‘cin’ function alone is not recommended, as cin reads unbounded input. |
| #include <iostream>  void f() {  char buf[12];  std::cin > buf;  } |

|  |
| --- |
| **Compliant Code** |
| Using std::string instead of a buffer array is the simplest way to protect against buffer overflow. Read into string objects, then manipulate as needed. |
| #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):** Validate Input Data. The most common way the system accepts input from users is in the form of strings (think usernames and passwords). It is important to verify the buffer size, and correct storage and manipulation of string variables in every part of the 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 | Checks for:  -Use of dangerous standard function  -Missing null in string array  -Buffer overflow from incorrect string format specifier  Destination buffer overflow in string manipulation  Rule partially covered |

### Coding Standard 4

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **SQL Injection** | IDS00-J | Prevent SQL Injection, primarily through the sanitization and validation of user input, as well as checking input taken in from external sources (like subsystems, other APIs, etc.) |

|  |
| --- |
| **Noncompliant Code** |
| The following code uses the keyword ‘username’ as an unsanitized parse for user input, which opens the application up for SQL Injection. |
| 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** |
| Use of parametric query with a ? character as a placeholder prevents SQL Injection |
| 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):** Validate Input Data. SQL Injection attacks most commonly occur in areas of the application where the user needs to log in or verify their identity. Verifying the type of information being passed into the system can help mitigate the potential for these kinds of 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 | 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** | MEM50-CPP | Do not access freed memory. Pointer arithmetic, especially concerning the usage and freeing up of memory within a program, is a leading cause of memory leak in applications. Dangling pointers such as this can result in security vulernatiblites. |

|  |
| --- |
| **Noncompliant Code** |
| Pointer s is dereferenced after it has been deallocated, leaving the code open to exploits. |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // …  delete s;  // …  s->f();  } |

|  |
| --- |
| **Compliant Code** |
| Dynamically allocate memory that is not deallocated until it is no longer required. |
| #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):** Keep It Simple. Memory management, especially in complex systems, can lead to a variety of errors including buffer overflow and the like. These errors may open the system up to vulnerabilities. Manage memory effectively, in simple ways, to ensure proper upkeep of the application. |

**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, but does not catch all violations of this rule |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |

### Coding Standard 6

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Assertions** | DCL03-C | Use a static assertion to test the value of a constant expression. The basic assert() macro, while functional, has its limitations specifically at runtime and for applications embedded into servers. The use of static\_assert() provides error messages as string objects at compile time. |

|  |
| --- |
| **Noncompliant Code** |
| Use of assert() macro to check the property of memory-mapped structures. |
| #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):** Use Effective Quality Assurance Techniques. This includes proper use of testing software/code, including the assert() macro. Use the best tools available for your quality assurance testing, as this is a crucial step in ensuring the long-term functionality of your application. |

**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 | Checked by clang-tidy |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| CodeSonar | 6.0p0 | (customization) | User can implement a custom check that reports the uses of the assert() macro |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

### Coding Standard 7

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Exceptions** | ERR50-CPP | Do not abruptly terminate the program. Calling functions such as std::abort() or std::\_Exit() close the program without exit handlers and without calling destructors for objects. This may leave external resources at risk, and as such these should only be called for a critical error in the application. |

|  |
| --- |
| **Noncompliant Code** |
| The f() function is used as an exit handler, but may result in an error in exceptions being thrown. |
| #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 thrown by the throwing\_func() |
| #include <cstdlib>  void throwing\_func() noexecpt(false);  void f() {  try {  throwing\_func();  } catch(...) {  // Handle error  }  }  int main() {  if (0 != std::atexit(f)) {  // 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. Exception handling is part of good software design, whether you throw a specific error to the user or simply block the functionality altogether. |

**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 | Use of abort  Use of 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 | Close files when they are no longer needed. Properly managing pointers, and the objects (such as files) they point to, is essential for preventing memory leakage in the application. Make sure to close files properly if the program no longer needs access. |

|  |
| --- |
| **Noncompliant Code** |
| File is constructed and read into memory, but the program is terminated which does not call constructors. Therefore the file will not 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()) {  // Handle error  }  std::terminate();  } |

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

|  |
| --- |
| **Principles(s):** Keep It Simple. Systems that use, open, and manipulate multiple files at once can get complicated. Ensuring proper handling of files, and the pointers which address them in memory, is a security practice that should be implemented in all parts of the application. |

**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 invoke virtual functions from constructors/destructors. The order of construction starts with base classes and moves to derived classes, so calling a derived class function from the base function call is dangerous (i.e. resource management issues). |

|  |
| --- |
| **Noncompliant Code** |
| Base class attempts to use object resources through virtual function calls. |
| 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 to nonvirtual and private member functions instead of 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. The use of OOP principles in any application inevitably leads to complexity. However, making sure to use the best, most standard practices for OOP, such as when to use (and not use) virtual function calls, can help keep code clean and more manageable. |

**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 | Do not use std::rand() for generating pseudorandom numbers. Using the standard library function from the C library does not guarantee quality of random number sequence generated. Some sequences can be short and easily predicted. |

|  |
| --- |
| **Noncompliant Code** |
| Generate ID with rand() function. ID is predictable and has limited randomness, therefore opening application up to security vulnerability. |
| #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** |
| Use of C++ library <random> allows for better control over random number generation by breaking the process into two parts: algorithm providing random values, and distribution of said values via a density function. |
| #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. Developers should make sure to use the most up-to-date standard of coding, such as using library functions in C++ rather than some outdated, less secure functions 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 | Checks for use of vulnerable pseudo-random number generator (rule partially covered) |

## 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.

The first area in which automation needs to be implemented to enforce policies is in the “Verify and test” phase of pre-production. Automatically checking the dependencies used by an application with a predefined and updated list of dependency vulnerabilities can greatly reduce the number of potential risks when moving product into the production phase. The logging system of the “Monitor and detect” phase during production should also be automated; notifications/logs should be automatically generated should the product stray from the guidelines outlined in this document, with detailed log files clearly labeled for future action. In addition to automatically generated logging of events, the system should also have the ability to automatically mitigate, or at least shut down, attempts at breaching said guidelines.

## 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 | Encryption at rest refers to the encryption of data that is not currently being used. For example, if you have a database full of usernames and passwords, or other potentially sensitive information that isn’t in constant use by your system, it is important to ensure that data is encrypted and secure; all too often the data breaches that make headlines are due to the data being stored in plain text or without any sort of security. |
| Encryption at flight | Encryption in flight, or also commonly called “in transit” is the process of encrypting data that is incoming or outgoing from your system. Examples include browsing the public internet and sending emails. It is important that the data in the system is encrypted at all stages, including traveling to and from the system from external sources as well as between components/modules of the system. |
| Encryption in use | Encryption in use is, as expected, the encryption and security of data currently being accessed by a user. This means having a secure, encrypted environment where data is created, viewed, and edited. There are several ways data in use can be encrypted, but the prime example would be during password verification. The use of a hashing function to encrypt the password, and a similar decryption function to read, translate, and verify the information is an example of encryption in use. |

|  |  |
| --- | --- |
| 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 the user trying to access the system. Be they employees or customers, anyone logging in to access the system should have at least a username and password combo that can be verified through the system (preferably through an encrypted database of username/password combos). Ideally, every person accessing the system should hae some form of alternate authentication, like trusted devices or separate codes sent to verified points of contact. |
| Authorization | Authorization is the process of checking and granting permissions to a user *after* they’ve successfully been authenticated into the system. For example, administrative users should have access to change the database, run queries, and perform system-level commands as needed. They should also have the ability to add users to the system, whereas the general consumer/user-level should only have access to specific files and program (with the absolute minimum of privileges to view and run things). |
| Accounting | Accounting, also known as Auditing, is the process of logging and accountability for the digital well being of the system and its data. The simplest example of this is a log file; a file that records the activity on the system, such as data accessed or edited, and the user account associated with those changes. This is a valuable part of digital security, as in the case of unauthorized access or attacks to the system, developers have a (hopefully) clear understanding of who the attacker was, how they access the system, and a roadmap as to how best protect against future repeat attacks. |

**\***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 | 03/21/2021 | Milestone - Coding Standards | Sawyer Prestwood |  |
| 1.2 | 04/09/2021 | Project Complete - Security Standards | Sawyer Prestwood |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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