**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 | Using malicious code as input data is an easy injection method if not validated. Usually these inputs are formed into statements that access external databases and such, so code held within input is a simple way to get to sensitive data quickly. One of the ways we can prevent this is, for example with SQL, is to use prepared statements, which require the developer to declare all SQL statements up front, and therefore prevent attackers from injecting their own code. |
| 1. Heed Compiler Warnings | Compiler warnings are an automatically generated source of vulnerability and should be respected for their informativeness. Usually, if it’s warned, it needs to be fixed in order to comply with coding best practices. |
| 1. Architect and Design for Security Policies | Keeping security policies separate keeps the design secure by only allowing certain privileges that are required by certain code segments. Giving someone the keys to your house is not a good idea if they are just applying new paint to the outside, and giving someone access to root is a terrible idea in pretty much all situations similarly. |
| 1. Keep It Simple | Occam’s razor usually applies to code. Complexity costs more time and effort, but simplicity is easier to read and more direct. Of course, complexity is naturally going to be prone to more edge cases that break code, so it’s best to try to keep it as simple as possible. |
| 1. Default Deny | Much like number 3, access should be denied for everything but what the task entails. Keeping all doors locked except for the ones that are necessary keeps intruders from meddling where they shouldn’t be. |
| 1. Adhere to the Principle of Least Privilege | I feel like I’m repeating myself, so this is obviously an extremely important principle in security protocol. Again keep all access as restricted as possible and only open access for specific purposes that the task entails. |
| 1. Sanitize Data Sent to Other Systems | Much like validating input data, sanitizing the data sent between subsystems is essential to preventing attacks via injection and others. It is the job of the calling process to clean this data, as the receiver doesn’t have context in which to process the data. |
| 1. Practice Defense in Depth | Do not have only one line of defense. Even if the wall is well-secured, if any vulnerability exists and that wall gets penetrated, pretty much everything after that is easy to exploit. Keeping secure subsystems is pertinent to security, and so having protection at every level is much more difficult to attack. |
| * 1. Use Effective Quality Assurance Techniques | QA techniques are the crack-fillers on every programming project, and should be looked at professionally by penetration testers and the like. Robust software is essential before release, so taking this seriously is a must. |
| 1. Adopt a Secure Coding Standard | Conforming to company standards is pretty much mandatory because having everyone on the same standard makes it easier to collaborate with others. Having your own set of principles is great, too and shows initiative. |

## 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** | API-07-C | Enforce type safety.  Upon return, functions should guarantee that any object returned by the function, or any modified value referenced by a pointer argument, is a valid object of function return type or argument type. Otherwise, type errors can occur in the program. |

| **Noncompliant Code** |
| --- |
| The standard strncpy() function does not guarantee that the resulting string is null-terminated. If there is no null character in the first n characters of the source array, the result may not be null-terminated. |
| **char** \*source;  **char** a[NTBS\_SIZE];  /\* ... \*/  **if** (source) {  **char**\* b = **strncpy**(a, source, 5); // b == a  }  **else** {    /\* Handle null string condition \*/  } |

| **Compliant Code** |
| --- |
| The C11 Annex K strncpy\_s() function copies up to n characters from the source array to a destination array. If no null character was copied from the source array, the nth position in the destination array is set to a null character, guaranteeing that the resulting string is null-terminated. |
| **char** \*source;  **char** a[NTBS\_SIZE];  /\* ... \*/  **if** (source) {    errno\_t err = strncpy\_s(a, **sizeof**(a), source, 5);  **if** (err != 0) {      /\* Handle error \*/    }  }  **else** {    /\* Handle null string condition \*/  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques – Double check to make sure type errors are handled or prevented. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | 4 | 3 |

**Automation**

****

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | MSC-52-CPP | Value-returning functions must return a value from all exit paths |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |

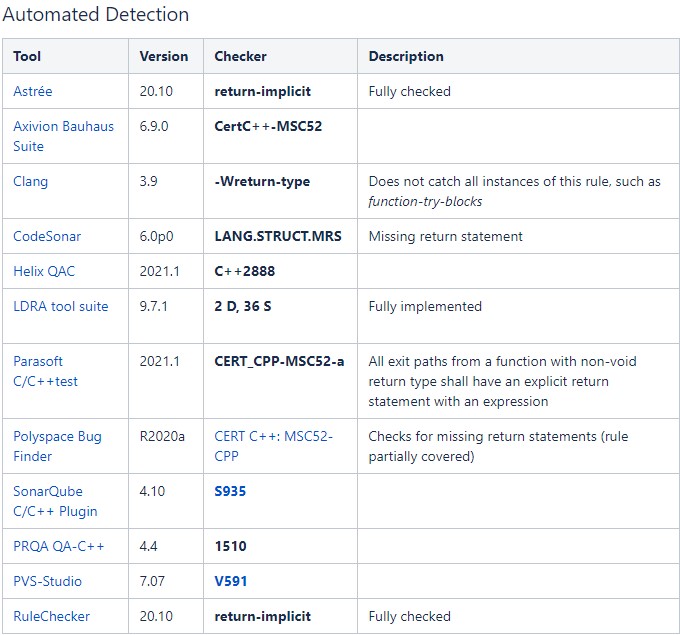
| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |

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

|  |
| --- |
| **Principles(s):** Adopt a secure coding standard – returning the correct data in all paths is a coding standard. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | 8 | 2 |

****

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | ERR-62-CPP | Detect errors when converting a string to a number.  Always explicitly check the error state of a conversion from string to a numeric value (or handle the related exception, if applicable) instead of assuming the conversion results in a valid value. This rule is in addition to ERR34-C. Detect errors when converting a string to a number, which bans the use of conversion functions that do not perform conversion validation such as std::atoi() and std::scanf() from the C Standard Library. |

| **Noncompliant Code** |
| --- |
| Multiple numeric values are converted from the standard input stream. However, if the text received from the standard input stream cannot be converted into a numeric value that can be represented by an int, the resulting value stored into the variables i and j may be unexpected. |
| **void** f() {  **int** i, j;    std::cin >> i >> j;    // ...  } |

| **Compliant Code** |
| --- |
| Exceptions are enabled so that any conversion failure results in an exception being thrown. However, this approach cannot distinguish between which values are valid and which values are invalid and must assume that all values are invalid. Both the badbit and failbit flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions. |
| **void** f() {  **int** i, j;      std::cin.exceptions(std::istream::failbit | std::istream::badbit);  **try** {      std::cin >> i >> j;      // ...    } **catch** (std::istream::failure &E) {      // Handle error    }  } |

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

|  |
| --- |
| **Principles(s):** Validate input data – Input should be validated to be sure it is in the correct format before executing further operations. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Mediun | Unlikely | Medium | 4 | 3 |

**Automation**

### 

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS-00-J | Prevent SQL Injection – Prevent attackers from using SQL queries |

| **Noncompliant Code** |
| --- |
| This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument username into the prepared statement. |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;    **class** Login {  **public** Connection getConnection() **throws** SQLException {      DriverManager.registerDriver(**new**              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  **return** DriverManager.getConnection(dbConnection);    }      String hashPassword(**char**[] password) {      // Create hash of password    }    **public** **void** doPrivilegedAction(      String username, **char**[] password    ) **throws** SQLException {      Connection connection = getConnection();  **if** (connection == **null**) {        // Handle error      }  **try** {        String pwd = hashPassword(password);        String sqlString = "select \* from db\_user where username=" +          username + " and password =" + pwd;        PreparedStatement stmt = connection.prepareStatement(sqlString);          ResultSet rs = stmt.executeQuery();  **if** (!rs.next()) {  **throw** **new** SecurityException("User name or password incorrect");        }          // Authenticated; proceed      } **finally** {  **try** {          connection.close();        } **catch** (SQLException x) {          // Forward to handler        }      }    }  } |

| **Compliant Code** |
| --- |
| 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):** Validate Input Data & Sanitize Data Sent To Other Systems – SQL Injection stems from input acceptance without validation or sanitation. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probably | Medium | 12 | 1 |

**Automation**

### 

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-52-CPP | Detect and handle memory allocation errors.  The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly. |

| **Noncompliant Code** |
| --- |
| an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

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

|  |
| --- |
| **Principles(s):** Use Effective QA Techniques – Handling memory issues is important to ensure a stable system |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 1 |

**Automation**

### 

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MET01-J | Never use assertions to validate method arguments.  Assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses assertions to validate arguments of a public method: |
| **public** **static** **int** getAbsAdd(**int** x, **int** y) {  **assert** x != Integer.MIN\_VALUE;  **assert** y != Integer.MIN\_VALUE;  **int** absX = Math.abs(x);  **int** absY = Math.abs(y);  **assert** (absX <= Integer.MAX\_VALUE - absY);  **return** absX + absY;  } |

| **Compliant Code** |
| --- |
| This compliant solution validates the method arguments by ensuring that values passed to Math.abs() exclude Integer.MIN\_VALUE and also by checking for integer overflow: |
| **public** **static** **int** getAbsAdd(**int** x, **int** y) {  **if** (x == Integer.MIN\_VALUE || y == Integer.MIN\_VALUE) {  **throw** **new** IllegalArgumentException();    }  **int** absX = Math.abs(x);  **int** absY = Math.abs(y);  **if** (absX > Integer.MAX\_VALUE - absY) {  **throw** **new** IllegalArgumentException();    }  **return** absX + absY;  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard – Using Assertions to validate data is against secure practice, as assertions are typically left out of final releases, therefore rendering them ineffective. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | 8 | 2 |

**Automation - N/A**

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR-51-CPP | Handle all exceptions.  All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard – Handling all Exceptions is standardized because if exceptions are left to a generalized handler, they are typically not handled in a meaningful way, somewhat like sweeping them under the rug. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | 4 | 3 |

**Automation**

### 

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Expressions** | EXP-30-C | Do not depend on the order of evaluation for side effects.  Evaluation of an expression may produce side effects. At specific points during execution, known as sequence points, all side effects of previous evaluations are complete, and no side effects of subsequent evaluations have yet taken place. Do not depend on the order of evaluation for side effects unless there is an intervening sequence point. |

| **Noncompliant Code** |
| --- |
| Programs cannot safely rely on the order of evaluation of operands between sequence points. In this noncompliant code example, i is evaluated twice without an intervening sequence point, so the behavior of the expression is undefined: |
| **void** func(**int** i, **int** \*b) {  **int** a = i + b[++i];  **printf**("%d, %d", a, i);  } |

| **Compliant Code** |
| --- |
| These examples are independent of the order of evaluation of the operands and can be interpreted in only one way: |
| **void** func(**int** i, **int** \*b) {  **int** a;    ++i;    a = i + b[i];  **printf**("%d, %d", a, i);  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard – When creating functions, explicitly creating an order of operations instead of leaving it up to an expected sequence is important to ensure desired effects are consistent. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | 8 | 2 |

**Automation**



### 

### 

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Object-Oriented Programming** | OOP-58-CPP | Copy operations must not mutate the source object.  When implementing a copy operator, do not mutate any externally observable members of the source object operand or globally accessible information. Externally observable members include, but are not limited to, members that participate in comparison or equality operations, members whose values are exposed via public APIs, and global variables. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the copy operations for A mutate the source operand by resetting its member variable m to 0. When std::fill() is called, the first element copied will have the original value of obj.m, 12, at which point obj.m is set to 0. The subsequent nine copies will all retain the value 0. |
| #include <algorithm>  #include <vector>    **class** A {  **mutable** **int** m;    **public**:    A() : m(0) {}  **explicit** A(**int** m) : m(m) {}      A(**const** A &other) : m(other.m) {      other.m = 0;    }      A& operator=(**const** A &other) {  **if** (&other != **this**) {        m = other.m;        other.m = 0;      }  **return** \***this**;    }    **int** get\_m() **const** { **return** m; }  };    **void** f() {    std::vector<A> v{10};    A obj(12);    std::fill(v.begin(), v.end(), obj);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the copy operations for A no longer mutate the source operand, ensuring that the vector contains equivalent copies of obj. Instead, A has been given move operations that perform the mutation when it is safe to do so. |
| #include <algorithm>  #include <vector>    **class** A {  **int** m;    **public**:    A() : m(0) {}  **explicit** A(**int** m) : m(m) {}      A(**const** A &other) : m(other.m) {}    A(A &&other) : m(other.m) { other.m = 0; }      A& operator=(**const** A &other) {  **if** (&other != **this**) {        m = other.m;      }  **return** \***this**;    }      A& operator=(A &&other) {      m = other.m;      other.m = 0;  **return** \***this**;    }    **int** get\_m() **const** { **return** m; }  };    **void** f() {    std::vector<A> v{10};    A obj(12);    std::fill(v.begin(), v.end(), obj);  } |

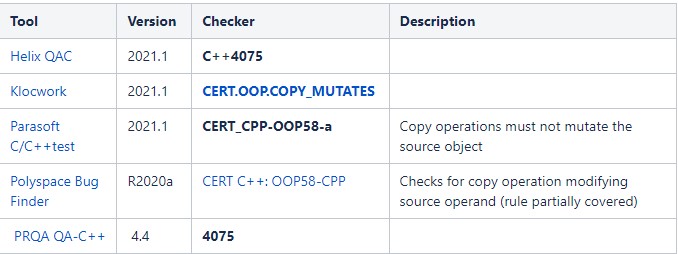
**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard & Practice QA Techniques – Mutating a source object when copying it can lead to unexpected behavior. Abiding by this standard will prevent unexpected errors. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | 9 | 2 |

**Automation**



### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Containers** | CTR-52-CPP | Guarantee that library functions do not overflow |

| **Noncompliant Code** |
| --- |
| STL containers can be subject to the same vulnerabilities as array data types. The std::copy() algorithm provides no inherent bounds checking and can lead to a buffer overflow. In this noncompliant code example, a vector of integers is copied from src to dest using std::copy(). Because std::copy() does nothing to expand the dest vector, the program will overflow the buffer on copying the first element. |
| #include <algorithm>  #include <vector>    **void** f(**const** std::vector<**int**> &src) {    std::vector<**int**> dest;    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

| **Compliant Code** |
| --- |
| The proper way to use std::copy() is to ensure the destination container can hold all the elements being copied to it. This compliant solution enlarges the capacity of the vector prior to the copy operation. |
| **void** f(**const** std::vector<**int**> &src) {    // Initialize dest with src.size() default-inserted elements    std::vector<**int**> dest(src.size());    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

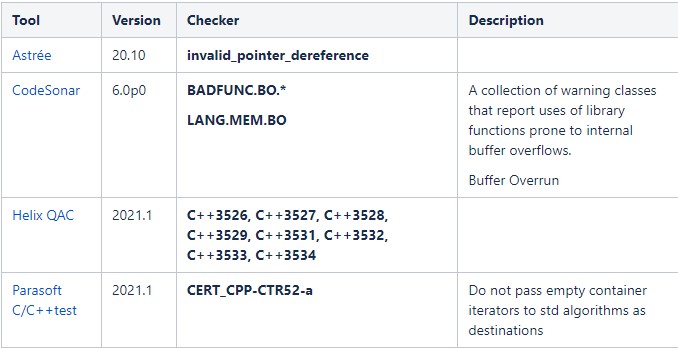
**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

|  |
| --- |
| **Principles(s):** Practice QA Techniques – Ensure that bounds are not overflowed even when using standard library functions. Much like a previous principle entailed explicitly specifying order of operations, explicitly creating allocated memory before using a library function to fill it will ensure consistent results that don’t overflow bounds. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 1 |

**Automation**



## 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.

Assess and Plan – This is a manual and creative process that cannot be automated

Design – Using automated tools inside of an IDE to enforce test-driven development such as JUnit for Java projects will ensure tests for all cases have been created. This is still dependent on manual write-ups such as UML diagrams for specificity.

Build – Automated repositories such as BitBucket allow developers to work on code together in a secured setting. Regulatory mechanisms such as privilege-controlled pushes will ensure a stable build is in development as well as test-builds.

Verify and Test – Using automated tools that scan for vulnerabilities more than a typical IDE is capable of is mandatory in implementing automated DevSecOps. There are too many to name and are application-specific but they are out there and need to be used for this to work.

Transition and health check – Penetration testers still use automated tools in order to scan for vulnerabilities. There are tons for specific applications. This still needs creativity in order to apply those tools to certain threat-models.

Monitor and Detect – Third-party software exists for this purpose in common household computers and company computers. Anti-malware software is prevalent and effective. Automation can also be implemented in the form of log-collection. This can be done by appending executed commands in an application to a log file to be reviewed or scanned automatically for threats, and creating an automatic alert system for the log results.

Respond – Manual intervention is applicable, not not mandatory in all cases. Automatic backups should always be regularly scheduled in case of a necessity for rollback. Attack-blocks can also be done automatically, depending on the attack detected via the alert system. Turning off services should be carefully considered when creating an automatic handler, as it affects users more drastically.

Maintain and Stabilize – Continually assessing current security measures, as well as continually threat-modeling should be an ongoing creative practice. If an attack is successful, assure a rollback is executed, and review log files and other output mechanisms for attack methods, and patch them in updates. Again, this is an ongoing process that should be done on a regular basis after initial measures have been completed.

## 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 |
| --- | --- | --- | --- | --- | --- |
| API-07-C | Medium | Unlikely | Medium | 4 | 3 |
| MSC-52-CPP | Medium | Probable | Medium | 8 | 2 |
| ERR-62-CPP | Medium | Unlikely | Medium | 4 | 3 |
| IDS-00-J | High | Probable | Medium | 12 | 1 |
| MEM-52-CPP | High | Likely | Medium | 18 | 1 |
| MET01-J | Medium | Probable | Medium | 8 | 2 |
| ERR-51-CPP | Low | Probable | Medium | 4 | 3 |
| EXP-30-C | Medium | Probable | Medium | 8 | 2 |
| OOP-58-CPP | Low | Unlikely | Low | 9 | 2 |
| CTR-52-CPP | High | Likely | Medium | 18 | 1 |

## 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 in rest is a way of storing information in a secure format, on a long-term storage media. This is done with a tool to encrypt data before it is stored. The tool is used as a medium between at-rest and at-flight transfer. It’s stored in this manner to prevent any onlookers from obtaining that data if elevated privilege allowed them access, either by a hack or even by an employee. Keeping data secure and practicing defense in depth is mandatory for data security in all states. |
| Encryption at flight | There are ways to intercept packets of data sent to and from systems over the internet. If those packets hold encrypted data, there’s not much that a hacker could do with them. Encrypting data before it’s sent over the internet prevents attackers from stealing the contents of the packets and using them for malicious intent. Encrypted packet necessity needs to be applied to sensitive information, such as bank login information being sent to/from a client to/from a server. |
| Encryption in use | As either a client or attacker is using data on their end, it should also be encrypted. This defense-in-depth application prevents potential client-side threats from accessing data on the user-end, and also prevents malicious users from abusing data sent to them. An example of this might be similar to the encryption at rest state, where a database admin with elevated privilege can use the data to perform certain operations, but should not be allowed to view the data in a non-encrypted state. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the practice of verifying that a user is actually who they say they are. Much like a license and registration are used to verify that a driver actually owns the vehicle they are driving, authentication methods such as usernames, passwords, and even Two-Factor Authentication methods such as text or email authentication are all ways to confirm that a user is not an impersonator. |
| Authorization | This is all about privileged access. A user must have a certain privilege in order to perform certain operations. For example, in order to keep a server safe from malicious intent, a regular user should not be granted root access to the server, which would allow them to do whatever they wish. A user with higher privilege might also not be allowed root access, but should be allowed read and write privileges to an encrypted database. The hierarchy of privilege follows the security principle, “Default deny”, which means that a user should only be allowed privilege that a specific authorized action requires, and blocking anything else. |
| Accounting | This involves logging of actions taken by a user. Any commands that are sent are logged and related to the particular user that performed those actions. This is so that nothing can slip under the radar unnoticed. Accounting for all actions is somewhat like the process of a customer renting a book from a library. Details are logged and assigned to the customer, so that they are accountable for the book they rented, in the same way that a user is accountable for the data they send to and from a server. That way, if the book is never returned (technically stolen), and similarly if data sent is in the form of an SQL query as malicious intent, the responsible party is known and actions can be taken to respond to the situation. |

**\***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.

| **Principle** | **Standard** | **Justification for connection** |
| --- | --- | --- |
| ValidateInput Data | ERR-62-CPP:  Detect errors when converting a string to a number | Input validation that require a certain data type such as a number is a requirement for the function to work properly. Invalid input will not be able to convert to an integer, thus breaking the functionality of the code. |
| ValidateInput Data | IDS-00-J:  Prevent SQL Injection – Prevent attackers from using SQL queries | Validating that an input does not include an SQL query is a mandatory requirement for secure systems connected to databases, such as login pages. Detecting and handling SQL query attempts from input fields helps to prevent database access from attackers. |
| Heed Compiler Warnings | MSC-52-CPP:  Value-returning functions must return a value from all exit paths | This type of standard is usually detected by the compiler, as all exit paths should be required to return a specified value. Not returning a value from a function practically renders the callee inert to the caller if it the caller is meant to use the returned value in some way. |
| Architect and Design for Security Policies | MET01-J:  Never use assertions to validate method arguments. | The architecture of validation should never rely on assertions for a release build. Assertions should only be used for testing purposes. The design should never rely on them to handle it in a real scenario. Often, assertions are not included in release builds, and therefore the system would then contain holes in logic. |
| Sanitize Data Sent to Other Systems | API-07-C:  Enforce Type-safety | Type-safety is an important thing to validate before sending it to an endpoint, because if the endpoint receives the wrong type, a type error will occur. Usually, this should be taken care of within the function that returns the type (the callee). |
| Sanitize Data Sent to Other Systems | ERR-62-CPP:  Detect errors when converting a string to a number | Conversion of data-types is another threat to type-safety if the types are non-convertible. Handling this error before sending it to another system is mandatory, as the system is expecting a certain data-type. |
| Sanitize Data Sent to Other Systems | IDS-00-J:  Prevent SQL Injection – Prevent attackers from using SQL queries | Rendering an SQL injection ineffective before sending data to another server such as an SQL database is an effective sanitation method for disallowing queried access. |
| Use Effective Quality Assurance Techniques | MEM-52-CPP:  Detect and handle memory allocation errors. | Avoiding memory allocation errors is a QA technique that protects against more than one threat, including system stability, and overflow injection as examples. Making sure that memory is allocated for all data recipients is part of the process that must not be overlooked. |
| Adopt a Secure Coding Standard | MSC-52-CPP:  Value-returning functions must return a value from all exit paths | It’s a standard requirement to return values from value-returning functions, in order to ensure code sanity / error-prevention. |
| Adopt a Secure Coding Standard | MET01-J:  Never use assertions to validate method arguments. | Standardizing checks for validation of method arguments does not include using assertions. Assertions are for Unit Testing, and cannot be the method of validation for a release build. |
| Adopt a Secure Coding Standard | ERR-51-CPP:  Handle all exceptions | Exceptions are a great way to keep a program stable in the case of errors, however, that tool should not be abused for its simplification. Handling all thrown errors properly is important to reroute a sequence. If an exception is left to a generalized catch-all statement, a sequence will simply not crash the program that instant, but it may behave unexpectedly later. All exceptions that are thrown should have their own corresponding catch block in order to keep a program on a stable track. The goal is to reroute an error, not to dismiss it. |
| Adopt a Secure Coding Standard | EXP-30-C:  Do not depend on the order of evaluation for side effects. | Explicit order of evaluation is going to behave as expected rather than relying on a presumed order. Programs may behave unexpectedly if left to its own order. Therefore standardizing the order in which operations are completed is key to consistent outcome. |

**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 |  |
| 1.0 | 06/10/2021 | Filled-out template initial | Trevor Peck | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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