**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 | To ensures any data that enters a system abides by specific rules and constraints, input of any kind must be validated. By validating the input, whether it be user input, files, etc., the data can be sure to it interacts safely with the underlying system or program and prevents any unexpected behavior. This security check prevents outside data from being intentionally or unintentionally used in a malicious or unexpected way, maintaining the integrity of the system and data itself. |
| 1. Heed Compiler Warnings | Compiler warnings can act as a form of early bug detection and if acted on early, can prevent expensive and costly patches or updates to fix an issue after deployment. The cost of addressing software issues increases dramatically the later it is addressed in the software development lifecycle. Compiler warnings can identify a broad range of issues related to memory safety, overflows, or memory management. Heeding these warnings ensure a potential defect is promptly addressed at the most opportune stage in the development lifecycle. |
| 1. Architect and Design for Security Policies | DevSecOps should be strongly considered. Incorporating Security Policies into the software architecture and design process ensures security is incorporating into a system inherently and prevents future attempts of fitting security around a system or project. Including security into the architecture and design process ensures the system itself, related policies, and team decisions are aligned. |
| 1. Keep It Simple | Simplicity in code makes it easier to read, maintain, and debug. Using well established algorithms and coding practices ensures a uniform approach to the development process and potentially reduces the complexity of security that is needed. Simple code is less likely to encounter unintended behavior or vulnerabilities and ensures clarity. |
| 1. Default Deny | Access in general should be denied by default unless otherwise granted to prevent the chances of unauthorized access to a part of a system, data, or resources. This reduces the entry points to a system and its vulnerabilities. This approach provides a zero-trust environment, which supports authorized access only. |
| 1. Adhere to the Principle of Least Privilege | Any process, user, or module should only be able to access the resources necessary to successfully complete it. By ensuring that this process is only accessing the required resources, the chances of unintentional behaviors from that process or person are dramatically reduced. This principle can apply to user privileges accessing a system or processes operating to complete a task. |
| 1. Sanitize Data Sent to Other Systems | When data is sent to other systems, sanitizing it helps ensure the receiving system is not prone to injecting attacks. Sanitizing this data prior to it being passed over to other parts of a system acts as another stage of validation and sanitization that removes potentially harmful characters. Additionally, data sanitization ensures compliance with policies protecting sensitive information, security breaches, and reduces the likelihood of data leakage. |
| 1. Practice Defense in Depth | Proper and effective security involves utilizing multiple layers of security that overlap and compensate weaknesses or openings of other layers used. Strong security policies ensure an efficient and thorough response to security incidents and overlapping security protocols and products ensure the system and its components are well protected. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance techniques for testing and code review are essential to identifying vulnerabilities earlier in the software development lifecycle and to ensure code behaves as intended. Integration testing, unit testing, and other form of Quality Assurance can be used to ensure a system is stable, secure, and operates as intended while also defining boundaries or identifying inefficiencies. |
| 1. Adopt a Secure Coding Standard | A secure coding standard can provide the building blocks or blueprints for ensuring a system is developed securely. These standards promote consistency in the development process while preventing security hazards. Whether the standard be related to a specific language, security protocol, or practices, it promotes a unified and cohesive development process from the start. |

### 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-CPP-001] | Obey the “One-Definition” rule. Any non-inline types, entities, or functions declared across files or translational units must abide by one definition. If a class is declared in one header-file as a specific data type, it cannot be declared elsewhere as a different data type. |

| **Noncompliant Code** |
| --- |
| This example shows a class being declared across two separate files. Although each of the files declare the class nearly the same, the data type in which it’s declared is not. |
| // file1.cpp  Struct S {    int a;  };    // file2.cpp  class S {  public:    int a;  }; |

| **Compliant Code** |
| --- |
| Instead of declaring the class multiple times across any translational units, the class can instead be included. This allows the same definition to be available and accessible in any translation units. In other words, the object can be declared in a header file and included in any others. |
| // headerfile.h  Struct S {    int a;  };    // file1.cpp  #include S.h  // file2.cpp  #include S.h |

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

| **Principles(s):**  10. Adopt a secure coding standard: The One Definition Rule is an important and relatively simple standard that can be achieved by adopting a secure coding standard that is understood and shared across a team.  4. Keep it simple: Simplicity can ensure there are not overly complicated methods and variables that can result in poor code maintenance and ensure declarations are singular.  3. Architect and Design for Security Policies: This ties into Adopting a Secure Coding Standard, but developing policies to ensure a team takes a uniform approach is important for designing policies for a team. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL60-a | A class, union or enum name (including qualification, if any) shall be a unique identifier |
| SonarQube | 8.9.9 | Duplicate Definition Checker | Identify usages of the same type or duplicate blocks of code |
| LDRA tool suite | 9.7.1 | 286 S, 287 S | Fully Implemented; Detecting duplicate definitions of the same type and code quality review. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [EXP-CPP-001] | Do not read uninitialized memory. Any local variables that are read prior to being initialized will use an indeterminate value that may result in unexpected behavior. |

| **Noncompliant Code** |
| --- |
| A function declares an array named *myArray*, but does not initialize it with anything. When this function is called on or if *myArray* is used before any data has been assigned to it, the program will lead to unexpected behavior. |
| Void printArray() {  int myArray[4];  For (int i = 0; i<4; ++1) {  Std::cout << myArray[i] << std::endl;  }  } |

| **Compliant Code** |
| --- |
| A function declares an array that is initialized with a default value before it is read or needs to print its value in the program or function. |
| Void printArray() {  Int myArray[4] = {0};  For (int i = 0; i<4; ++1) {  Std::cout << myArray[i] << std::endl;  }  } |

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

| **Principles(s):**  10. Adopt a secure coding standard: Uniform coding standards develop the framework that an organization can abide by, and initializing memory is a crucial and likely inclusion with these standards.  7. Sanitize data sent to other systems: This coding standard can bring implication to a single system, but can also cause issues with any other systems it interacts with, so it is crucial that data be properly sanitized prior to being sent.  4. Keep it simple: Avoiding overly complicated code can be an effective way to prevent uninitialized memory and maintain code.  2. Heed Compiler Warnings: Compilers can be an important front-line support that can catch this sort of issue early on in the development process, making it easier to catch early and reduce the cost of remediation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Critical (P12) | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Front-end compiler for diagnosing code errors. May not detect uninitialized values from heap memory. |
| Astrée | 22.10 | uninitialized-read | Partially checked.  Static Analyzer for a variety of runtime errors and diagnostic reporting |
| LDRA tool suite | 9.7.1 | 53 D | Partially implemented.  Identifies some non-initialized variables or pointers. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STR-CPP-001] | Guarantee that storage for strings has sufficient space for character data and the null terminator. This emphasizes the importance of allocating enough memory for a string to ensure a buffer overflow does not occur. |

| **Noncompliant Code** |
| --- |
| The ‘sample’ buffer can only contain 20 bytes, meaning strcpy will overflow the buffer with the sample text provided. In other words, this does not safely handle the string. |
| char sample[20];  strcpy(sample, “TestingABCDEFG123456890HIJKLMN”); |

| **Compliant Code** |
| --- |
| Instead of relying on a bounded array, this uses std::string to prevent buffer overflows while also preventing the input from being cut off. |
| void f() {d  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):**  1. Validate Input Data: Checking to make sure that a variable of some kind has the appropriate amount of storage space for a string directly ties into validating the input data that can be used for that storage.  8. Practice Defense in Depth: Considering that buffer overflows are a common attack vector, this coding standard ties directly into the practice of defense in depth by ensuring secure coding is practiced.  10. Adopt a secure coding standard: As mention with previous examples, adopting secure coding standards ensure that common forms of attacks are considered throughout the development process, which include ensuring something like a buffer overflow cannot occur. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 22.10 | stream-input-char-array | No space for null terminator  Buffer overrun  Type overrun |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the 'char' buffer to store input from 'std::cin' |
| Polyspace Bug Finder | R2024a | CERT C++: STR50-CPP | Use of dangerous standard function  Missing null in string array  Buffer overflow from incorrect string format specifier  Destination buffer overflow in string manipulation Insufficient destination buffer size  Rule partially covered. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [IDS-CPP-001] | Prevent SQL Injection by using prepared statement and parameterization when handling any user input for interacting with a database. |

| **Noncompliant Code** |
| --- |
| This code calls on the getUsername() and getPassword() functions to take the input from the user for logging in. However, when creating the sqlString, it simply passes the user-provided username and password directly into the SQL string that will be passed to the database, allowing the user to perform SQL Injection if they so choose. |
| std::string username = getUsername();  std::string password = getPassword();  std::string sqlString = “SELECT \* FROM db\_user WHERE username = '" +  username + "' AND password = '" + pwd + "'";  Statement statement = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString); |

| **Compliant Code** |
| --- |
| A prepared statement is created, which allows the SQL statement to escape and be properly parameterized. |
| sql::Connection \*con;  sql::PreparedStatement \*prep\_stmt  prep\_stmt = con->prepareStatement(“SELECT \* FROM db\_user WHERE username =? AND password =?”;  prep\_stmt-> setString(1, username);  prep\_stmt->setString(2, password);  prep\_stmt->execute(); |

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

| **Principles(s):**  1. Validate Input Data: SQL Injections are a direct consequence of improperly validating input data. It is crucial for a system to practice secure input validation measures to prevent SQL injections.  7. Sanitize Data Sent to Other systems: This is another crucial principle for preventing SQL injection attacks. Just as it is important for a system to validate input, it is important for it to sanitize data sent to another system.  8. Practice Defense Depth. This security approach ensures that there are multiple layers that address a potential SQL injection attack. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and Security Errors |
| Parasoft JTest | 2024.1 | CERT.IDS00.TDSQL | Protection against SQL Injection |
| SonarQube | 9.9 | S2077, S3649 | Executing SQL queries is security sensitive and can provide Taint Analysis in the developer edition. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-CPP-001] | Properly deallocate dynamically allocated resources. When attempting to free dynamically allocated memory using ‘delete’, ‘delete[]()’, etc., make sure to use the appropriate option depending on what allocation method was used. |

| **Noncompliant Code** |
| --- |
| This function attempts to allocate memory for the i1 and i2 variables using *new*, but if either fails, it will attempt to call on *delete* to deallocate the memory. However, because i1 and i2 were never initialized prior to the allocation method, the delete() operator may be passed an unknown value in i2. |
| void f() {  int \*i1, \*i2;  try {  i1 = new int;  i2 = new int;  } catch (std::bad\_alloc &) {  delete i1;  delete i2;  }  } |

| **Compliant Code** |
| --- |
| Instead of not initializes the pointer variable i1 and i2, they are initialized with nullptr, allowing the delete operator to proceed as intended in the event that memory cannot be allocated for i1 or i2 new(). |
| void f() {  int \*i1 = nullptr, \*i2 = nullptr;  try {  i1 = new int;  i2 = new int;  } catch (std::bad\_alloc &) {  delete i1;  delete i2;  }  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: Adopting a coding standard is crucial to properly allocating and deallocating memory as needed and understanding when to use which. These standards ensure a team is following a widely accepted and reviewed method of handling this process.  9. Use Effective Quality Control: Thorough QA testing can and should catch when memory is not properly allocated or deallocated, making this a principle that should be abided by when navigating this standard.  8. Practice Defense in Depth: Memory leaks and preventing them are an important security concern. Defense in Depth inherently not only addresses the underlying cause of this problem, but also introduces other layers of security to address an incident if it occurs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| Polyspace Bug Finder | R2024a | CERT C++: MEM51-CPP | Checks for:  Invalid Deletion of point Invalid Free of pointer  Deallocation of previously deallocated pointer |
| CodeSonar | 8.1p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable Double free  Type mismatch Leak |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [DCL-002-CPP] | Use a static assertion to test the value of a constant expression. If conditions that can be evaluated at compile time, it’s better to use static assertions to improve code quality and performance. However, runtime assertions can and should be used for conditions that can only be checked during runtime. |

| **Noncompliant Code** |
| --- |
| An assert is used to test is used, but it is a runtime assertion to check if myData.data is null. Because of this, it has to be placed into a function and executed, and likely has some distance between the actual structure it is referencing and requires the actual code to be run for anything to work with the assertion – if it does at all. |
| struct Data {  int myNum;  char\* myData;  };  int main() {  Data data = {42, nullptr};  assert(myData.data != nullptr);  return 0;  } |

| **Compliant Code** |
| --- |
| Instead of using a runtime assertion, this using a static\_assert to diagnose potential issues at compilation, meaning there is no runtime cost and would provide an actual compilation error if an issue were to occur instead of a potential silent error or runtime error. |
| struct Data {  int myNum;  char\* myData;  };  static\_assert(sizeof(struct Data) == sizeof(int) + sizeof(char\*),  “Potential size error”); |

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

| **Principles(s):**  8. Use Effective Quality Assurance Techniques: Assertions are inherent to quality assurance and the testing phase of the lifecycle of software development. Effective QA techniques include understanding when to use static assertions and runtime assertions to prevent unintentionally applying the wrong form to a scenario that calls for one.  2. Heed Compiler Warnings: Using static assertions can be an effective way to identify errors that occur during compilation. Heeding the compiler warnings directly relate to the use of assertions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully Implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully Implemented |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [ERR-001-CPP] | Handle all exceptions. Anytime an exception is thrown, a matching handler needs to exist to facilitate the handling of the try block in which the exception occurred. The lack of such a handler results in the terminate() operation to occur, which can result in external resources associated with the unwound stack being left in an indeterminate state instead. |

| **Noncompliant Code** |
| --- |
| An exception is thrown from the function ‘someFunction’, there is another function ‘anotherFunction’ that calls on ‘someFunction’. In the main function, ‘anotherFunction’ is called on, but none of these actually includes code to catch and handle the thrown exception. |
| void someFunction() noexcept(false);    void anotherFunction() {  someFunction();  }    int main() {  anotherFunction();  } |

| **Compliant Code** |
| --- |
| Instead of only including a function that throws an exception without any other function actually handling and catching the exception, the main function now includes a try-catch block to attempt calling ‘anotherFunction()’ and handling any exceptions thrown. |
| void someFunction() noexcept(false);    void anotherFunction() {  someFunction();  }    int main() {  try {  anotherFunction();  }  catch(…){  // Handle error  }  } |

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

| **Principles(s):**  3. Architect and Design for Security Principles: The appropriate handling of exceptions is a core part of designing and architecting with security principles in mind, in addition to just implementing quality code. Exceptions are designed to handle events that are not quite what was intended, meaning it is crucial for software to have an appropriate handler for each. This ties into incorporating security principles into the design and architecture of the system.  10. Adopt a Secure Coding Standard: Implementing an handler for each instance of an exception can be accommodated through adopting a coding standard. The two pair with architect and design for security principles as well.  9. Use Effective Quality Assurance Techniques: Exceptions are an inherent part of quality control, as the handling of them is essential for a system to appropriately manage scenarios in which an exception occurs. Solid quality assurance techniques will ensure that any exception is appropriately handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++test | 2023.1 | 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. |
| Klocwork | 2024.2 | MISRA.CATCH.ALL | [Insert text.] |
| Polyspace Bug Finder | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Reserved Identifiers | [DCL-002-CPP] | Do not declare or define a reserved identifier. There are several reserved names with specific rules that need to be abided by. Users should avoid defining identifiers as it may result in unexpected behavior. |

| **Noncompliant Code** |
| --- |
| C++ uses specific words as keywords, meaning they carry a specific purpose and should not be defined by the user. In this example, the keyword “class” is used, but has also been defined by the developer. When a special keyword is used as a macro that the user defines, unexpected behavior may occur. |
| #define class 20  Int main() {  int myNum = class;  return 0;  } |

| **Compliant Code** |
| --- |
| Instead of using a special keyword, the user should define their own variable. Instead of using “class” in the user defined marco, this example demonstrates using a different and valid word instead. |
| #define STUDENTS 20  Int main() {  int myNum = STUDENTS;  return 0;  } |

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

| **Principles(s):** 10. Adopt a secure coding standard: Coding standards often abide by specific naming conventions and standards that prevent unintentionally naming something with a reserved identifier.  9. Use Effective Quality Assurance Techniques: Part of quality assurance involves various forms of code testing, which includes manual coding review. By thoroughly reviewing code, especially earlier on in the development process, can effectively catch and correct instances of a reserved identified being used in code.  4. Keep it Simple: Many variables can easily be named in such a way that does not need to use a reserved identifier. By keeping variable names simple, code can avoid problematic scenarios of being named after reserved identifiers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | reserved-identifier | Partially Checked. A static analyzer that can use abstract implementation for runtime errors. |
| Clang | 3.9 | -Wreserved-id-macro  -Wuser-defined-literals | This flag does not  catch all instances of this rule, such as redefining reserved names. |
| Polyspace Bug Finder | R2024a | [Insert CERT C++: DCL51-CPP.] | Checks for redefinitions of reserved identifiers (rule partially covered) |
| LDRA Tool Suite | 9.7.1 | 86 S, 218 S, 219 S, 580 S | Fully Implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Container Indices | [CTR-001-CPP] | Guarantee that container indices and iterators are within the valid range |

| **Noncompliant Code** |
| --- |
| In this example, a function is passed two parameters that can be influencer from untrusted sources. While this does perform a check on the range to ensure one of the parameters doesn’t exceed the upper limit, it doesn’t perform the same chck to ensure it doesn’t exceed the lower limit. Instead of the ‘pos’ parameter being declared as something else, it’s passed as a signed int, meaning there is the potential for it to be negative. |
| void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {    if (pos >= tableSize) {      return;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| Instead of leaving the position parameter as an int, it can be declared as std::size\_t. This prevents it from accepting a negative argument. |
| void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {    if (pos >= tableSize) {      return;    }    table[pos] = value;  } |

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

| **Principles(s):**  1. Validate Input Data: When it comes to container indices, it’s crucial to ensure that any data being passed to them fits and abides by a certain accepted criterion, which involves validating that input data. Ensuring that any data used is properly validated can be an effective principle to ensure this standard is properly maintained.  9. Use Effective Quality Assurance Techniques: Testing is an important aspect of security. It ensures that a program behaves in the way it is meant to, identifies potential edge cases, and addresses those security gaps. This principle can be a valuable way to identify and address when container indices and iterators may not be within a valid or accepted range. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference | An abstract static analyzer that can identify runtime errors that may occur |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range. |
| CodeSonar | 8.1p0 | LANG.MEM.BO, LANG.MEM.BU, LANG.MEM.TO, LANG.MEM.TU, LANG.MEM.TBA, LANG.STRUCT.PBB, LANG.STRUCT.PPE, LANG.STRUCT.PARITH | Buffer overrun, Buffer underrun, Type overrun, Type underrun, Tainted buffer access, Pointer before beginning of object, Pointer past end of object, Pointer Arithmetic |
| Helix QAC | 2024.2 | C++3139, C++3140 DF2891 | C++ static code analyzer |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object’s Memory Lifetime | [EXP-002-CPP] | Do not access an object outside of its lifetime. When an object’s memory has been deallocated, has been destructed, or reused, it should not be used or there can be unexpected behavior. Pointers referencing local variables whose lifetime has already ended can result in issues. |

| **Noncompliant Code** |
| --- |
| A function Is used to return a pointer to the address of a local variable ‘i’, However, when this function, returnPointer(), is used, the local variable is destroyed. The second function is simply a function that is passed a parameter that is a pointer ‘i’. Lastly, anotherFunction() calls on returnPointer() and creates a pointer to a local variable that is no longer valid, so when someFunction() uses i, it no longer points to the memory location that is still valid. |
| int\* returnPointer() {  int i = 15;  return &I;  }  void someFunction(int\* i);  void anotherFunction() {  int\* i = returnPointer();  someFunction(i);  } |

| **Compliant Code** |
| --- |
| Instead of declaring i with an automatic storage duration, it can be declared with a static duration instead. This would extend the lifetime of ‘i’ so it can be used in other functions. |
| int\* returnPointer() {  static int i = 15;  return &I;  }  void someFunction(int\* i);  void anotherFunction() {  int\* i = returnPointer();  someFunction(i);  } |

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

| **Principles(s):**  10. Adopt a secure coding standard: Accessing an object outside of its lifetime can result in undefined behavior, which is something that is heavily considered with secure coding standards. Similarly to strong coding practices, this principle encourages consistent code that strongly reduces the likelihood of this incident occurring.  8. Practice Defense in Depth: A major principle of defense in depth is multiple layers of security that start with the raw data itself, handling that data, and radiates outward to policy. This principle pairs with adopting a secure coding standard, as they each overlap and aim to mitigate the chance of this issue happening. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<>; Can also check if a variable is stored and never accessed. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | Do not use resources that have been freed  The address of an object with automatic storage shall not be returned from a function  The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist |
| Code Sonar | 8.1p0 | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation can be a powerful and crucial tool when enforcement and complying with the security standards defined in this policy. While Green Pace may has a well-established DevOps process and infrastructure, it can be dramatically enhanced by incorporating the security standards and automation into their pipeline. Incorporating security into the software development lifecycle means implementing security throughout the process. It’s important to identify each of the phases used in the DevSecOps pipeline and acknowledge where automation can be incorporated.

Assessing and planning serves as the starting point for the DevSecOps pipeline, and it is at this stage that automation can be utilized to enhance the planning process. This is a stage where security policies can be accepted and addressed, but also a stage where potential threats can be modeled and understood. Automation tools such as OWASP ZAP and ThreatModeler both serve as powerful tools to assess a current system’s status, vulnerabilities, and develop a plan for what needs to be addressed. These forms of automation ensure proper input validation, appropriate handling of exceptions, and employing adopting secure coding standards.

The Design and Build stages can utilize automation in a variety of ways, but one of the most effective ways would be to incorporate IDE plugins and implement Infrastructure as Code tools. Some of these include Terraform, which would automate the deployment, or analysis tools such as SonarQube. Static analysis tools such as Clang can be used for real-time feedback on code as it is developed. The Verify and Test phase can extend from the automation incorporated into the Building stage, where static analysis tool can automate unit testing. Additionally, dependency checks can be automated to identify any vulnerabilities in libraries that are used in the project. However, there are powerful automation tools that can OWASP Dependency Check and Snyk that can be used to identify vulnerabilities in these libraries and dependencies. Compliance checks can be automated through the use of OpenSCAP as well. These further implement the security standards outlined in this document by practicing defense in depth.

The transition and health check phase and Monitor and detect phase can automated, but it depends heavily on the project and code used. For example, there are security configuration tools like AWS. Automation tools such as GitLab, and Jenkins can promote a Continuous Improve Continuous Development approach for deployment. Additionally, the monitoring and detection phase a huge opportunity to incorporate automation to adhere to the security standards defined here. There are powerful tools that can be used to detect issues, monitor logs, and respond to them accordingly. There are tools like Grafana and Kibana that provide powerful tools to store, search, and identify logs for a system that streamline the process and allow a team to identify potential issues.

Responding and Maintenance of the DevSevOps pipeline can also use automation. There are tools like Jira/JSM that provide a powerful tool for responding and reacting to issues if they arise. Automation tools exist that allow a company to incorporate Infrastructure-as-Code such as Ansible or Puppet that can be used to enable the management of the system’s infrastructure in the event an attack is experienced. Automation tools that provide version history management such as Git provide an easy way for a company to roll back their system. These enforce a Defense in Depth, Adopting a Secure Coding Standard, and Architect and Design for Security Policy standard when in the Respond and Maintain and stabilize phases.

It is crucial for automation and security principles to be incorporated throughout the DevSecOps pipeline to ensure these standards can be enforce and complied with. Implementing a CI/CD approach can have a dramatic effect on the security of the business and ensure security principles and standards can be enforced.

### 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-CPP-001 | High | Unlikely | High | Low (P3) | 3 |
| EXP-CPP-001 | High | Likely | Medium | Critical (P12) | 1 |
| STR-CPP-001 | High | Likely | Medium | Critical (P18) | 1 |
| IDS-CPP-001 | High | Likely | High | Critical (P18) | 1 |
| MEM-CPP-001 | High | Likely | Medium | Critical (P18) | 1 |
| DCL-002-CPP | Low | Unlikely | High | Low (P1) | 3 |
| ERR-001-CPP | Low | Probably | Medium | Medium (P4) | 3 |
| DCL-002-CPP | Low | Unlikely | Low | Low (P3) | 3 |
| CTR-001-CPP | High | Likely | High | High (P9) | 2 |
| EXP-002-CPP | High | Probable | High | Medium (P6) | 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 at rest | Data at rest refers to when any sort of data is not being transmitted or used, and instead is being stored. This can be in the form of data being stored into a hard drive, database, etc., so encryption at rest pertains to securing that data when it is stored to ensure it is protected from being accessed or read by any authorized users. As such, any stored data, especially pertaining to clients, identifiable information, and any internal information, should be encrypted with industry standards, such as AES-256. Access to decryption keys shall be strictly monitored and restricted to authorized users with a default of denying access. |
| Encryption in flight | When data is being sent over a network or somewhere, whether that be for storage purposes or for using, it should be encrypted as to ensure it is protected from any attempts to intercept or access it from unauthorized users. As such, any data in such a state shall be encrypted using protocols such as HTTPS, Transport Layer Security (TLS), or employ data integrity checks using SHA-256. This is to prevent any sort of unauthorized eavesdropping or intercepting of the data during transmission, especially when using a public network. This applies to any sensitive, client, internal, or financial data in transit. |
| Encryption in use | Data that is being processed, prepared, and used by the system or application is considered data is use and shall require encryption in this state. When a data is being accessed, read, or otherwise processed, it is considered “in use” and should employ encryption practices such as Partially/Fully homomorphic encryption or secure multi-party computation. Since data typically needs to be converted to plain text when in use, it can be made vulnerable. To prevent such vulnerabilities, these encryption methods shall be used to prevent any sort of unauthorized access when data is being prepared, read, or otherwise accessed, especially for client, internal, or otherwise sensitive data in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is an important factor in maintaining the security of a system, as it ensures and validates a particular user or system trying to access any particular resources or aspects of the system. This step of authentication is crucial to implement as it acts as a barrier that prevents the wrong users from interaction with a system. As such, authentication should be implemented through industry accepted means such as OAuth 2.0 in addition to multi-factor authentication when attempting to log into any accounts or access internal tools, especially when involving accounts with authority to access sensitive data or resources. This is to ensure that even if an authorized account is accessed, the attacker will still need to use MFA to access any resources from which they could do harm with. |
| Authorization | User accounts will have varying permissions depending on their role, needs, or attributes when it comes to accessing various parts of a system, making changes, or otherwise modifying a system’s resources. This process of authorization is to restrict what users could potentially access sensitive information, view parts of a system, or otherwise make changes to it. As such, standard forms of authorization protocols shall be implemented across any users accessing the system and abide by secure coding standards, such as Role-Based Access Control. Authorization practices should abide by the default deny principle and adhere to the principle of least privilege. This is to ensure that any account that should not be able to access various parts of the platform do not unintentionally gain access to them and to restrict the attack surface from which a threat actor could exploit. |
| Accounting | Monitoring and tracking any access, or attempts to access, the system or its resources should be exercised to ensure all activity is accounted for. Logging activity in this manner can be an effective way to not only monitor potential security vulnerabilities, but also provide the ability to respond to suspicious activities. As such, it is crucial to monitor user’s actions, system activity, changes to a database through any available tools, such as Grafana or Kibana. Logs should be regularly audited and reviewed to identify suspicious activity. By incorporating strong accounting practices, this policy will provide an effective route to take when attempting to detect and respond to any security incidents as well as identifying suspicious activity before an incident occurs, resulting in a security breach. |

**\***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 |  |
| 2.0 | 07/15/2024 | Added security principles and coding standards with acceptable and unacceptable code examples. | Ryan Hoskins | [Insert text.] |
| 3.0 | 08/09/2024 | Added Risk Assessment, encryption, automation, Triple A, and Summary of Risk Assessment. | Ryan Hoskins | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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