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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Test to ensure all input data is correct and complete before processing. This prevents buffer overflows, injections and errors created by malicious input. |
| 1. Heed Compiler Warnings | Warnings that notify the developer of wrong code and vulnerabilities. Using these can prevent incorrect practices and bugs within the code earlier in the process. |
| 1. Architect and Design for Security Policies | This allows for policies and security requirements to be implemented within the design phase of development, reducing the vulnerabilities in the system. |
| 1. Keep It Simple | Keep It Simple means keeping the code simple so that it is easier to understand and maintain, reducing the number of errors in the code/design. |
| 1. Default Deny | By default, block all access unless permitted. This reduces attacks by preventing unauthorized use. |
| 1. Adhere to the Principle of Least Privilege | Gives and systems the minimal permission allowed to perform tasks necessary. By limiting use and permission, less malicious attacks are performed potentially. |
| 1. Sanitize Data Sent to Other Systems | Ensure the data leaving the system is correct and clean. This prevents injection into other systems and data corruption of other systems. |
| 1. Practice Defense in Depth | Use multiple layers of security so that if one layer fails, there is another to back it up. This allows for a better, secure system in case of attack. |
| 1. Use Effective Quality Assurance Techniques | Testing, audits and code reviews are incorporated to detect or fix errors. This prevents vulnerabilities early on and makes for better reliability. |
| 1. Adopt a Secure Coding Standard | Follow best standards for writing code to reduce vulnerabilities and meet security needs. |

### 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** | [STD-001-CPP] | Obey the one definition rule |

| **Noncompliant Code** |
| --- |
| Two files are creating a class with the same name but the definition of the class is not the same. This makes it so the complier does not see them as the same type. |
| // a.cpp  **struct** S {  **int** a;  };    // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| By creating a header file, the object is defined. By using the same header in both files, the class is defined the same way in both files. |
| // S.h  **struct** S {  **int** a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** 2,4,10.  2. Heed Complier Warnings. This help detects one definition rule violations early on.  4. Keep it Simple. This reduces the chance of duplicates and keeps code organized.  10. Adopt a Secure Coding Practice. This makes sure definitions are consistent across. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Type compatibility, definition-duplicate, undefined-extern, external-file-spreading, type-file-spreading | Partially checks for ODR violations with type and definition tracking |
| LDRA Tool Suite | 9.7.1 | 286 S, 287 S | Checks for ODR compliance through translation units, fully implemented |
| Polyspcace Bug Finder | R2024b | CERT C++: DCL60-CPP | Partially implements ODR compliance by checking for inline constraints not respected |
| RuleChecker | 22.10 | Type-compatibility, definition-duplicate, undefined-extern, undefined-extern-pure-virtual, external-file-spreading, type-file-spreading | Partially checks for multiple definitions and ODR issues |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| A pointer is used to call a member function before it actually points to a valid object. This will lead to undefined behavior. |
| **struct** S {  **void** mem\_fn();  };    **void** f() {    S \*s;    s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| Memory is correctly allocated for the object before calling the function. The behavior is correctly defined because the pointer is pointing to a correct object. |
| **struct** S {  **void** mem\_fn();  };    **void** f() {    S \*s = **new** S;    s->mem\_fn();  **delete** s;  } |

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

| **Principles(s):**1,2,8  1. Validate Input. This ensures the input is valid to check if the object is still valid.  2. Heed complier warnings. These can detect dangling pointers and warn about lifetime issues.  8. Practice defense in depth. Using runtime checks and adding layers of security prevents vulnerabilities like use-after-lifetime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wdangling-initializer-test | Detects lifetime issues within std::initializer\_list<> |
| RuleChecker | 22.10 | Return-reference-local | Partially checks for returning references to locals |
| PVS Studio | 7.38 | V758, V1041, V1099 | Flags returning and using references to destroyed/temporary objects |
| CodeSonar | 9.1p0 | IO.UAC, ALLOC.UAF | Checks for use-after-close and use-after-free |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| The std::string is created using std::getenv() but if the variable “TMP” does not exist, std::getenv() will return nullptr. This will lead to undefined behavior. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| Now, the code checks first if the result from std::getenv() was nullptr, if not, the string becomes initialized with value. If it is null, an empty string is used. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** 1,2,4,8,9  1. Validate Input. This principle will ensure null-checking before using pointers.  2. Heed Compiler Warnings. Many compiler warnings will generate warnings based on null pointers being passed to a std::string. Resolving these errors before runtime is better.  4. Keep it Simple. Keeping the string logic simple decreases the chance of using null pointers and errors.  8. Practice Defense in Depth. Check pointers even when the risk is low, this principle enforces that to add extra security.  9. Use Effective Quality Assurance Techniques. Runtime analysis tools will detect null dereferences. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | LANG.MEM.NPD | Detects null pointer dereference |
| Parasoft C/C++ test | 2024.2 | CERT\_CPP-STR51-a | Test to avoid null pointer referencing |
| Polyspace Bug Finder | R2024b | CERT C++:STR51-CPP | Partially checks for string operators on null pointer |
| Helix QAC | 2025.2 | DF4770-DF4774 | Detects dereferencing null or uninitialized pointers |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| The program takes a username and password and inserts into SQL string. If the input has malicious data, the structure of the query can change. |
| uName = getRequestString(“username”);  uPass = getRequestString(“userpassword”);  sql = “SELECT \* FROM users WHERE Name = " + uName + " AND Pass = " + uPass + ” |

| **Compliant Code** |
| --- |
| This way does not involve direct insertion of input into the SQL query. It uses a statement, treating the input as data and not code. This helps prevent SQL injection. |
| PreparedStatement pStmt = PreparedStatement ();  std::cin >> username;  std::cin >> userpassword;  sql = “SELECT \* FROM Users WHERE Name = %s AND PASS = %s;”, username, userpassword}; |

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

| **Principles(s):** 1,7,8,9  1. Validate input. SQL injection is often caused by not validating input and entering the query. This can be prevented through principle 1.  7. Sanitize data sent to other systems. Cleaning the queries before sending it to other systems can prevent SQL injections.  8. Practice defense in depth. Adding additional security layers even after validation will help ensure no SQL injection occurs.  9. Use effective quality assurance techniques. Penetration testing/automation allows for protection verification against SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Fully checks for untrusted strings within query |
| Klockwork | 2025.2 | SV.DATA.DB, SV.SQL, SV.DATA.DBSOURCE | A comprehensive SQL injection test |
| Parasoft JTest | 2024.2 | CERT.IDS00.TDSQL | Protection against SQL injection |
| SonarQube | 9.9 | S2077, S3649 | SQL injection rule that query should not have dynamic strings |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| The program creates an object, s, using new and deletes it. The memory pointer has already been released so the program can crash or malicious code could be implemented. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In the corrected version, the object is used before being deleted and the pointer is not accessed after the memory has been freed. |
| #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):** 1,2,6,8  1. Validate Input. This ensures that inputs are valid and not using deallocated memory.  2. Heed Compiler Warnings. This warns if the pointer is dereferenced after being freed and detects occurrences where the same memory is deallocated multiple times.  6. Adhere to the Principle of Least Privilege. By limiting the lifetime of pointers, use-after-free bugs attacks are diminished.  8. Practice Defense in Depth. Use of pointers, memory sanitizer and multiple layers of security allows more checking of memory protections. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | ALLOC.UAF | Detection of use after free |
| Coverity | 7.5.0 | USE\_AFTER\_FREE | Detects use of freed memory |
| Parasoft C/C++ Test | 2024.2 | CERT\_CPP-MEM50-a | Do not use any resources that have been freed |
| Astree | 22.10 | Dangling\_pointer\_use | Partial detection of any dangling pointer usage |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| By using the assert method, the structure size is being checked to make sure it matches. Assert() is only ran in debug builds, meaning it can be skipped at runtime. |
| #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** |
| --- |
| Rather then relying on assert(), this way uses #if which is used at complier time. The #error will stop the run with an error. This way, the size structure is checked. |
| **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):** 2,3,4,9  2. Heed Compiler Warnings. This warns if static\_assert conditions are incorrect.  3. Architect and Design for Security Policies. Using static\_assert enforces the type sizes and layouts. This makes sure the architecture adheres to the standards which reduces runtime errors.  4. Keep it Simple. Static\_assert makes compile times self-documenting, resulting in less errors during runtime. It also makes the code maintainable and easy to follow.  9. Use Effective Quality Assurance Techniques. This ensures that conditions are being met before the application runs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Misc-static-assert | Detect any opportunity to use static\_assert |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented for static assertion |
| Code Sonar | 9.1p0 | Customizable | Can detect use of assert() macros |
| Security reviewer | 6.02 | C13, C14, C15, C52, C129 | Detects assertion misuse, fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| There is no code to catch exception, the program terminates abruptly without properly finishing up. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| Using a try-catch block, exceptions are caught and the program can clean up without terminating abruptly. |
| **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):** 3,8,9  3. Architect and Design for Security Policies. Designing effective exception handling makes sure there are less failures/errors.  8. Practice Defense in Depth. Making sure exceptions are caught is important to prevent crashes and data leaks, multiple layers of handlers is best to have less exposure.  9. Use Effective Quality Assurance Techniques. Using static analysis testing and runtime testing, exception safety can be confirmed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | LANG.STRUCT.UCTCH, PARSE.MBDH | Detects any uncaught or hidden exceptions |
| Klockwork | 2025.2 | MISRA.CATCH.ALL | Includes a detailed catch coverage |
| Polyspace Bug Finder | R2024b | CERT C++: ERR51-CPP | Partially checks for any unhandled exceptions |
| Astree | 22.10 | Main-function-catch-all, early-catch-all | Partial detection of exceptions thrown |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-008-CPP] | [Guarantee that container indices and iterators are within the valid range](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range) |

| **Noncompliant Code** |
| --- |
| Using the function insert\_in\_table(), it takes pos and value as integers. It also does not check if pos is negative which means negative values can flow through. This leads to undefined behavior. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, **int** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| This version allows for checking of negative values and checks that the index is zero or positive values. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, std::**size\_t** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

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

| **Principles(s):** 1,2,4,6,7,8,9  1.Validate Input. Container indices can come from untrusted sources, including user input, this principle ensures that any index is checked before use. This prevents logic errors and memory corruption.  2. Heed Compiler Warnings. Compilers can detect the use of iterators after that have uninitialized variables. This principle can resolve the warning before code is compiled.  4. Keep it Simple. When logic is too complex, more errors and risk occur. Keeping it simple allows for reduction of potential flaws.  6. Least Privilege. With containers, this principle will only give functions access to what they need specifically and nothing else, providing more security.  7. Sanitize Data Sent to Other Systems. Invalid indices can inject bad values. Validating indices protects the external and local systems.  8. Defense in Depth. This places multiple layers of security likes compile assertions and runtime checks. So, if another fails, this will catch the errors.  9. Use Effective Quality Assurance Techniques. Using testing like static analysis testing can catch out of bound access. Using this principle will integrate automatic testing into the standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2025.2 | C++3139, C++3140, DF2891 | Covers out of range indexing |
| Parasoft C/C++ test | 2024.2 | CERT\_CPP-CTR50-a | Ensures that containers are within the correct range |
| Polyspace Bug Finder | R2024b | CERTC++:CTR50-CPP | Checks for arrays that are out of bounds, array access that has tainted index and dereferences with tainted offset, partially covered |
| PVS-Studio | 7.38 | V781 | Index boundary violation |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-009-CPP] | [Write constructor member initializers in the canonical order](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order) |

| **Noncompliant Code** |
| --- |
| The order of member variables being declared is different in the class. This reads as an undefined value and results in incorrect behavior. |
| **class** C {  **int** dependsOnSomeVal;  **int** someVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| The member variables are now reordered in class declaration, making it match the order of the constructor list. |
| **class** C {  **int** someVal;  **int** dependsOnSomeVal;    **public**:    C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** 2,4,10  2. Heed Compiler Warnings. Incorrect order often shows in warnings about undefined behavior, so this principle is important to incorporate to show if order is incorrect.  4. Keep it Simple. Declaring and initializing members in an organized way will lessen the chance of errors.  10. Adopt a Secure Coding Standard. Coding standards will define the correct ordering to avoid memory issues, therefore it is important to include this principle for this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wreorder | Complier warning for the initializer order |
| Klockwork | 2025.2 | CERT.OOP.CTOIR.INIT\_ORDER | Mapping to rule for writing constructor methods in order |
| SonarQube C/C++Plugin | 4.10 | S3229 | Flags any incorrect order in the initializer list |
| Helix QAC | 2025.2 | C++4053 | Checks the order of initializer list |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-010-CPP] | Use correct integer precisions |

| **Noncompliant Code** |
| --- |
| This function is calculating 2 raised to the power of the input “exp”. It checks if exp is too small compared to the number of bits in the int. If exp is too large, an exception occurs. |
| #include <limits.h>    unsigned **int** pow2(unsigned **int** **exp**) {  **if** (**exp** >= **sizeof**(unsigned **int**) \* CHAR\_BIT) {      /\* Handle error \*/    }  **return** 1 << **exp**;  } |

| **Compliant Code** |
| --- |
| This way uses a function popcount() to count the number of bits that are set to 1. This way it can know the bit precision of any integer type (signed or unsigned). |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  **size\_t** popcount(uintmax\_t num) {  **size\_t** precision = 0;  **while** (num != 0) {  **if** (num % 2 == 1) {        precision++;      }      num >>= 1;    }  **return** precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s):** 1,2,4,6,8,9  1. Validate Input. Validating input ensures no invalid shifts occur. This standard adheres to integer operations which are based around input values.  2. Heed Compiler Warnings. Compiler warnings can warn of overflows and invalid shifts, using this principle ensures these are caught early on.  4. Keep it Simple. Misusing integer types can increase risk, as well as using complex integer types. This principle allows for clear and simple code with standard integer types.  6. Least Privilege. Use the smallest integer type necessary. If a value will never exceed the max amount, use uint8\_t to decrease the level of threat.  8. Defense in Depth. Combining warnings and testing will make sure there is safe integer use.  9. Use Effective Quality Assurance Techniques. This principle can help avoid unsafe arithmetic and shift errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | Overflow checks | Detects shifts that are larger than bit-width |
| CodeSonar | 9.1p0 | LANG.ARITH.BIGSHIFT | Checks if shift amount exceeds bit width |
| Helix QAC | 2025.2 | C0582, C++3115 | Detects out of range operations |
| Polyspace Bug Finder | R2024b | CERT C: Rule INT35-C | Partially covered, checks for shift validation |

### 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.

During the planning and development process, developers should use static analysis tools to gain feedback on any violations for the coding standards. These tests can include Parasoft C/C++ Test and Clang. Next, in the build and test phase, static code analysis should be applied to the continuous integration process. Tools like SonarQube, CodeSonar or Helix QAC should be used. These tools are made to automatically scan code against any CERT rules. Therefore, any violations that are detected, the build should be flagged and marked as failed. Before deploying the code, the static analysis should be reviewed further by the team. Once deployed, runtime analysis tools, like Parasoft Insure ++, can be used to detect memory issues and undefined behavior. This is in case the static analysis tool missed these. Logging and monitoring tools should be implemented to ensure there are no runtime exceptions.

Using automation during every phase of development allows for accurate coding standards and better security. It reduces vulnerabilities, checks for error and code quality, and shortens the feedback loop. Integrating automation tools into the process provides industry-best standards for development.

### 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 |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Low | High | 2 |
| STD-003-CPP | High | Likely | Low | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Low | High | 2 |
| STD-006-CPP | Low | Unlikely | Low | Low | 3 |
| STD-007-CPP | Low | Likely | Low | High | 2 |
| STD-008-CPP | High | Unlikely | Low | High | 2 |
| STD-009-CPP | Medium | Unlikely | Low | High | 2 |
| STD-010-CPP | Low | Unlikely | Low | Low | 3 |

### 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 | Encryption at rest is encryption applied to data stored on a physical/virtual machine. It is used to prevent breaches that involve physical theft or inappropriate access to a system. Is it used when long-term data is stored. |
| Encryption in flight | Encryption in flight is applied to any data in transmission over networks. An example is between client and server. It is used to protect data from being tampered with during transmission and required when sensitive data is sent over networks. |
| Encryption in use | Encryption in use applies to data during memory processing. It protects data from exposure during active processing. It is good to use in high security environments where data should be kept confidential. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identify of the user on the system. This can be incorporated through passwords, multi-factor authentications or biometrics. It is important to prevent any access that is not authorized and used all systems that contain sensitive information/data. |
| Authorization | Authorization is when access is granted or denied to specific users or rights. This can be through policies or restrictions and RBAC/ABAC. It also requires approval for addition of new users. Authorization makes sure users can only access the data if they have permission. It is required for the least-privilege standard and to minimize risk. |
| Accounting | Accounting is the act of tracking user activities while recording them. It is implemented through audits, logging and recording systems. It keeps an audit of any new users added and logs files that were accessed or changes to the database. This is required for systems that need to comply with regulations. |

**\***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 |