**Green Pace Developer: Security Policy Guide**



# 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 | In an effort to protect against buffer overflow abuse, injection attacks, or other vulnerabilities related to input, user input must be validated. This reduces the risk of malicious pieces of data causing security breaches or system failures. |
| 1. Heed Compiler Warnings | Compiler warnings should be investigated as they can indicate potential security issues, such as type mismatches or uninitialized variables. |
| 1. Architect and Design for Security Policies | Security needs to be implemented into the software architecture at the start of development to proactively address potential threats. Adopting this approach lowers the risk that new vulnerabilities will be introduced later in development. |
| 1. Keep It Simple | Simple code and software designs should be utilized in order to make it easier to review and maintain for other developers. Complex systems are much harder to audit, can hide small, subtle bugs, and can make security vulnerabilities harder to detect. |
| 1. Default Deny | All access should be denied unless explicitly granted. Doing this lowers the attack surface, which is a word for the total possible unauthorized entry points into a system. |
| 1. Adhere to the Principle of Least Privilege | Processes and users must only have access to the necessary data and resources required for them to adequately perform their functions. Adherence to this principle can reduce the magnitude of security breaches in the unfortunate circumstances that they occur. |
| 1. Sanitize Data Sent to Other Systems | Data that is sent from out of our systems must be effectively sanitized to reduce the risk of a continued propagation of security breaches. Properly sanitizing all of our data will ensure communications coming from our systems are safe. |
| 1. Practice Defense in Depth | Security must be implemented in several layers to mitigate risk and provide a failsafe in the event that a layer fails. Utilizing DiD ensures other defenses are still in place even if one may fail. |
| 1. Use Effective Quality Assurance Techniques | Regular code testing, which includes reviews of our code and the use of automated testing, should be implemented to detect security problems early. The use of proper QA processes can help ensure vulnerabilities are caught prior to release. |
| 1. Adopt a Secure Coding Standard | Adherence to secure coding standards like the CERT C++ Coding Standard goes a long way toward helping to reduce common vulnerabilities, like buffer overflows or SQL injection attacks. Following these standards will ensure consistent security practices are followed by everyone. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | In C++, if a reference type is qualified with const or volatile, the resulting code is ill-formed due to the fact that references inherently act as const-qualified types. Attempting to apply these qualifiers to references is redundant and prone to errors. ([CERT](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile#:~:text=C%2B%2B%20does%20not%20allow%20you,references%20as%20being%20const%20qualified.)) |

| **Noncompliant Code** |
| --- |
| This noncompliant example declares p to be a reference to a const-qualified char. The modification of p following this declaration makes the program ill-formed. |
| #include <iostream>  Void referencingConst (char c) {  const char &p = c;  p = ‘p’; // Error: this is a read-only variable and is not // assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| The const qualifier is taken away in this compliant code. |
| #include <iostream>  Void properReferencing (char c) {  char &p = c;  p = ‘p’; // This assignment works properly after removing const  std::cout << c << std::endl;  } |

| **Principles(s):**  **3. Architect and Design for Security Policies:** Creating code without redundant qualifiers will ensure the architecture remains secure and avoids unintended behavior.  **4. Keep It Simple:** Maintaining simple code by avoiding any unnecessary qualifiers will make it easier for other developers and yourself to read, maintain, and understand, which reduces the risk of introducing new vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 |  |
| Helix QAC | 2024.2 | C++0014 |  |
| Klocwork | 2023.1 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE |  |
| Polyspace Bug Finder | R2024a | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule is fully covered |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Unsigned integer operations can result in wrapping when a value outside the range that can be represented with that data type is produced. This causes it to “wrap” around modulo the maximum value plus one. Integer wrapping should especially be avoided in security-critical code or memory indexing operations where a buffer overflow or other vulnerability could occur from the wrap. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap)) |

| **Noncompliant Code** |
| --- |
| This noncompliant example could result in an integer wrap during the addition of ui\_a and ui\_b. If this behavior is not planned for, the resulting value could be used to allocate insufficient memory for a subsequent operation or some other exploitable vulnerability. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum = ui\_a + ui\_b;  /\* … \*/  } |

| **Compliant Code** |
| --- |
| This solution utilizes a precondition test of the operands to ensure there is no possibility of unsigned wrap. |
| #include <limits.h>  void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum;  if (UINT\_MAX – ui\_a < ui\_b) {  /\* Handle error \*/  } else {  usum = ui\_a + ui\_b;  }  } |

| **Principles(s):**  **1. Validate Input Data:** Ensuring that integer operations do not wrap is an essential part of input validation.  **8. Practice Defense in Depth:** Adding checks for unsigned integer operations is another layer of defense that can be added to prevent unintended behaviors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 24.04 | integer-overflow | Implemented |
| Cppcheck Premium | 24.9.0 | premium-cert-int30-c | Partially implemented |
| Helix QAC | 2024.2 | C2910, C3383, C3384, C3385, C3386  C++2910  DF2911, DF2912, DF2913 |  |
| TrustInSoft Analyzer | 1.38 | unsigned overflow | Exhaustively verified |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Modifying string literals in C++ results in an undefined behavior due to the fact that the literals are typically stored in read-only memory. Attempting to change them would cause an access violation. String literals should only be assigned to pointers or arrays marked as const to prevent accidental modifications. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/STR30-C.+Do+not+attempt+to+modify+string+literals)) |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the char pointer str is initialized to the address of a string literal. The subsequent attempt to modify the string literal results in an undefined behavior. |
| char \*str = “string literal”;  str[0] = ‘S’; |

| **Compliant Code** |
| --- |
| This solution creates a copy of the string literal and allocates it to the character array str. This allows the string stored in str to be modified safely. |
| char str[] = “string literal”;  str[0] = ‘S’; |

| **Principles(s):**  **1. Validate Input Data:** By ensuring string literals are verified to not be modified, unpredictable behavior is avoided and the correct handling of input data is enforced.  **6. Adhere to the Principle of Least Privilege:** Limiting the ability to modify string literals aligns with the idea of restricting access to functionality and resources to only what is necessary for proper operation.  **9. Use Effective Quality Assurance Techniques:** Utilzing proper QA processes will help to ensure that situations where string literals have unintended modifications are identified early and resolved without much issue. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | LIKELY | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | string-literal-modification write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| PC-lint Plus | 1.4 | 489, 1776 | Partially supported |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | String data passed to complex subsystems have the possibility of containing special characters, which could trigger commands or actions and result in a software vulnerability. All string data passed should be sanitized to ensure the resulting string is innocuous in its interpretation context. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems)) |

| **Noncompliant Code** |
| --- |
| This noncompliant example shows an application that inputs an email address to a buffer and then uses that string as an argument in a call to system(), which could allow for an SQL injection attack. |
| sprintf(buffer, “/bin/mail %s < /tmp/email”, addr);  system(buffer); |

| **Compliant Code** |
| --- |
| This compliant solution utilizes a whitelist to define a list of acceptable characters and remove any that are not accepted. |
| Static char ok\_chars[] = “abcdefghijklmnopqrstuvwxyz”  “ABCDEFGHIJKLMNOPQRSTUVWXYZ”  “1234567890\_-.@”;  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 = ‘\_’;  } |

| **Principles(s):**  **1. Validate Input Data:** Ensures that data passed within complex subsystems is checked and verified.  **7. Sanitize Data Sent to Other Systems:** Sanitization aids in ensuring data sent out does not contain potentially harmful content.  **8. Practice Defense in Depth:** The implementation of multiple sanitization and verification layers provides redundancy in defending against SQL injection attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | Medium | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Klocwork | 2024.2 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Clear sensitive information stored in reusable resources** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Sensitive data that is stored in reusable resources may be leaked to a less privileged user or attacker if not properly cleaned. The manner in which the information can be properly cleaned varies depending on what the resource type is and the platform its running on. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/MEM03-C.+Clear+sensitive+information+stored+in+reusable+resources)) |

| **Noncompliant Code** |
| --- |
| In this example, sensitive data stored in the dynamically allocated memory referenced by secret is copied to the buffer, new\_secret, which is processed and eventually deallocated by utilizing a call to free(). Since the memory is not clear, there is a possibility it can be reallocated to another section of the program, which could lead to the data in new\_secret being leaked. |
| char \*secret;  /\* Initialize secret to a null-terminated byte string,  of less than SIZE\_MAX chars \*/    size\_t size = strlen(secret);  char \*new\_secret;  new\_secret = (char \*)malloc(size+1);  if (!new\_secret) {  /\* Handle error \*/  }  strcpy(new\_secret, secret);    /\* Process new\_secret... \*/    free(new\_secret);  new\_secret = NULL; |

| **Compliant Code** |
| --- |
| This compliant solution clears the previously allocated space by filling it with ‘\0’ characters. |
| char \*secret;  /\* Initialize secret to a null-terminated byte string,  of less than SIZE\_MAX chars \*/    size\_t size = strlen(secret);  char \*new\_secret;  /\* Use calloc() to zero-out allocated space \*/  new\_secret = (char \*)calloc(size+1, sizeof(char));  if (!new\_secret) {  /\* Handle error \*/  }  strcpy(new\_secret, secret);    /\* Process new\_secret... \*/    /\* Sanitize memory \*/  memset\_s(new\_secret, '\0', size);  free(new\_secret);  new\_secret = NULL; |

| **Principles(s):**  **6. Adhere to the Principle of Least Privilege:** Clearing sensitive data ensures that no unauthorized process or user is able to access data that has no authorization for.  **8. Practice Defense in Depth:** Removing sensitive data from more exposed areas adds an additional layer of protection against data leaks by ensuring memory allocated for sensitive purposes cannot be misused. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | (customization) | Users can add a custom check for use of realloc() |
| Helix QAC | 2024.2 | C5010 |  |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |
| Parasoft C/C++ test | 2023.1 | CERT\_C-MEM03-a | Sensitive data should be cleared before being deallocated |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a powerful tool for finding and eliminating defects in software but have some limitations in that they incur a runtime overhead and because they call abort(). Because of this, assert() is useful for identifying incorrect assumptions and not for runtime error checking. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression)) |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses assert() to assert a property concerning a memory-mapped structure that is essential 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** |
| --- |
| This compliant solution shows a preprocessor conditional statement instead. |
| 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 |

| **Principles(s):**  **2. Heed Compiler Warnings:** Utilizing assertions during compile time will help catch issues early on, ensuring code performs as intended.  **9. Use Effective Quality Assurance Techniques:** Static assertions are an important quality assurance measure that can catch errors prior to runtime. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | HIGH | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions that are not caught result in std::terminate() being called and abnormally terminating the process, which could result in the stack not unwinding and creating a potential security vulnerability or resource leak. Catching all exceptions ensures the program can properly release its resources and terminate in a controlled manner. ([CERT](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions)) |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, exceptions thrown by throwing\_func() are not caught by either f() or main(). Because there is no matching handler for the exception, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| This compliant solution handles all exceptions, ensuring the stack is unwound and allows for proper management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

| **Principles(s):**  **3. Architect and Design for Security Policies:** Handling exceptions properly will ensure that software design properly and securely accommodates error conditions.  **8. Practice Defense in Depth:** Ensuring that all exceptions are caught provides an additional layer of security in that uncaught errors are prevented from causing crashing or leading to new vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ -ERR51 |  |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable catch |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Ensure that division and remainder operations do not result in divide-by-zero errors** |
| --- | --- | --- |
| Integers | [STD-008-CPP] | In C++, divide-by-zero errors result in undefined behavior that could result in a program crash or the creation of a security vulnerability. By ensuring all division and remainder operations have no possibility involving a zero divisor, these types of unpredictable behaviors can be prevented. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/INT33-C.+Ensure+that+division+and+remainder+operations+do+not+result+in+divide-by-zero+errors)) |

| **Noncompliant Code** |
| --- |
| This noncompliant example fails to prevent a divide-by-zero error during the division of the signed operands s\_a and s\_b. |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_a == LONG\_MIN) && (s\_b == -1)) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This solution tests the division operator to ensure there is no possibility of a divide-by-zero error. |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Principles(s):**  **1. Validate Input Data:** Ensuring inputs are valid by checking if a divide-by-zero is going to happen is essential to prevent errors.  **9. Use Effective Quality Assurance Techniques:** Proper input validation that prevents division by zero can be ensured through proper quality assurance testing. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | LIKELY | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | int-division-by-zero  int-modulo-by-zero | Fully checked |
| CodeSonar | 8.1p0 | LANG.ARITH.DIVZERO  LANG.ARITH.FDIVZERO | Division by zero  Float division by zero |
| Coverity | 2017.07 | DIVIDE\_BY\_ZERO | Fully implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-INT33-a | Avoid division by zero |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not use object representations to compare floating-point values** |
| --- | --- | --- |
| Floating-Point | [STD-009-CPP] | Comparing the object representations of floating-point values can result in incorrect conclusions regarding their equivalence due to the fact that values like positive zero and negative zero may have different representations but are considered equal, while NaN values could have identical representations but would not be equal. Utilizing equality operators like == and != ensures all comparisons are accurate by accounting for the special rules of floating-point arithmetic. ([CERT](https://wiki.sei.cmu.edu/confluence/display/c/FLP37-C.+Do+not+use+object+representations+to+compare+floating-point+values)) |

| **Noncompliant Code** |
| --- |
| This noncompliant example compares two structures with floating-point objects for equality, which could result in unintended behavior. |
| #include <stdbool.h>  #include <string.h>    struct S {  int i;  float f;  };    bool are\_equal(const struct S \*s1, const struct S \*s2) {  if (!s1 && !s2)  return true;  else if (!s1 || !s2)  return false;  return 0 == memcmp(s1, s2, sizeof(struct S));  } |

| **Compliant Code** |
| --- |
| This solution compares the floating-point structure members individually and with proper operators. |
| #include <stdbool.h>  #include <string.h>    struct S {  int i;  float f;  };    bool are\_equal(const struct S \*s1, const struct S \*s2) {  if (!s1 && !s2)  return true;  else if (!s1 || !s2)  return false;  return s1->i == s2->i &&  s1->f == s2->f;  } |

| **Principles(s):**  **4. Keep It Simple:** The utilization of proper comparison techniques for floating-point values avoids undefined behavior and unnecessary complexity.  **9. Use Effective Quality Assurance Techniques:** Using proper QA techniques will ensure value comparisons are handled properly and accurately, reducing the potential errors that would be caused if left untouched. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Memcmp-with-float | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FLP37 | Fully implemented |
| LDRA tool suite | 9.7.1 | 618 S | Enhanced Enforcement |
| TrustInSoft Analyzer | 1.38 |  | Exhaustively verified |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Write constructor member initializers in the canonical order** |
| --- | --- | --- |
| Object Oriented Programming | [STD-010-CPP] | The initialization order of base classes and member variables is determined by the order they were declared in the class, not by the order in the constructor’s initializer list. Writing constructor member initializers in the canonical order that matches the declaration order prevents both confusion and potential bugs by ensuring the code properly reflects the sequence of initialization. ([CERT](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP53-CPP.+Write+constructor+member+initializers+in+the+canonical+order)) |

| **Noncompliant Code** |
| --- |
| This noncompliant example attempts to initialize someVal and then dependsOnSomeVal to a value dependent on someVal. Because the declaration order of the member variables does not the same as the member initializer order, attempting to read someVal results in an unspecified value being stored into dependsOnSomeVal. |
| class C {  int dependsOnSomeVal;  int someVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| This solution changes the declaration of the member variables to ensure the dependency can be ordered properly in the constructor’s initializer list. |
| class C {  int someVal;  int dependsOnSomeVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Principles(s):**  **3. Architect and Design for Security Policies:** Ensuring that member initializers follow their canonical order aligns with secure coding standards and minimizes initialization errors.  **4. Keep It Simple:** Maintaining the canonical order of initializers helps simplify the code and prevents subtle bugs. This contributes to easier maintenance and an overall more secure system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Initializer-list-order | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CerrtC++-OOP53 |  |
| LDRA tool suite | 9.7.1 | 206 S | Fully implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP\_OOP53-a | List members in an initialization list in the order in which they are declared |

### 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.

**Pre-Production**

It is essential that automation is integrated throughout the DevSecOps process in a way that both enforces and ensures compliance with Green Pace’s secure development standards. Beginning with the assess and plan phase, automation should be utilized to analyze the threat landscape, regulatory changes, and other factors that could impact security requirements. Doing this helps in adjusting the current development backlog and ensuring new threats are found and planned for early. In the design phase, automated measures, such as input validation and automated security validation, can be used to adhere to secure design principles and apply best practices like ones from the Open Worldwide Application Security (OWASP). For the build phase, automation should be utilized to secure builds by ensuring that only trusted repositories are used. Dependencies should be verified using automated validation tools to help maintain a consistent build process at this time. Finally, the verify and test phase consists of using automated vulnerability scanning tools to identify vulnerabilities and perform compliance testing. All of this helps to make the final product free from vulnerabilities before being deployed.

**Production**

During the production phases of a project, automation is mainly used to maintain system security and respond quickly to ongoing incidents or suspicious activity. The first phase, transition and health check, involves employing automated deployment tools to assist with configuring security settings and validating the overall security of the system with penetration testing. During the monitor and detect phase, developers utilize automated monitoring tools, such as SIEM (Security Information and Event Management) systems, which are able to continuously record logs, analyze events as they occur, and detect in-progress intrusion attempts to ensure they are addressed in a timely manner. The respond phase sees automated mechanisms being used to block attacks, disable affected systems or services, and initiate rollback procedures to mitigate incidents. During the final phase of production, maintain and stabilize, the system is regularly assessed against current security baselines, with automation being used to monitor compliance and restore to secure states in the event that deviations are found after an attack. This comprehensive implementation of automation across the entire development pipeline ensures continuous security and minimizes the need for manual intervention.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Low | Low | 1 |
| STD-002-CPP | HIGH | LIKELY | HIGH | Medium | 2 |
| STD-003-CPP | Low | LIKELY | Low | Medium | 2 |
| STD-004-CPP | HIGH | LIKELY | Medium | HIGH | 1 |
| STD-005-CPP | Medium | Unlikely | HIGH | Low | 3 |
| STD-006-CPP | Low | Unlikely | HIGH | Low | 3 |
| STD-007-CPP | Low | PROBABLE | Medium | Low | 3 |
| STD-008-CPP | Low | LIKELY | Medium | Medium | 2 |
| STD-009-CPP | Low | Unlikely | Medium | Low | 3 |
| STD-010-CPP | Medium | Unlikely | Medium | Low | 3 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the process of encrypting data while it is in storage, whether on physical or virtual storage, like databases, file systems, and system backups. This policy ensures that all sensitive data, which includes user information or private credentials, is encrypted with a strong encryption algorithm. An algorithm such as AES-256 can aid in ensuring sensitive data is virtually uncrackable through brute force attacks and can maintain data confidentiality even in the event that physical devices are compromised. This policy applies whenever data is saved or archived to make sure it is secure even during long periods of inactivity. |
| Encryption in flight | Encryption in flight refers to the concept of encrypting data as it is being transmitted across networks, such as between the devices of clients or different internal components of an application. This policy necessitates the use of secure communication protocols to prevent data interception by unauthorized individuals. The purpose of using a secure protocol like TLS (Transport Layer Security) is to both maintain data confidentiality and integrity while it is in transit. This protects against man-in-the-middle attacks, where a third-party attacker can intercept and affect communications between two parties. Application of this policy should be done whenever data needs to move across networks, public or internal, ensuring it is protected from outside intrusion. |
| Encryption in use | Encryption in use is the practice of encrypting data while it is actively being processed, such as during computations. Applying this policy requires techniques like homomorphic encryption, which is a form of encryption that allows mathematical operations to be performed on the data without needing to decrypt it first. This policy’s purpose is to secure sensitive data against unauthorized memory access or data scraping attacks by ensuring sensitive information is protected while processing it. This policy applies to all critical operations where sensitive data has the possibility of being loaded into memory in order to ensure data privacy and compliance with security standards and regulations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of users, systems, or services that are attempting to access resources or data. The application of this policy enforces secure practices like multi-factor authentication (MFA) for all users and services that can access sensitive data or systems. This policy serves to provide a robust layer of security that ensures only legitimate users can access critical resources and sensitive data, in turn reducing the risk of unauthorized access. Authentication applies whenever users log in, new users are added, or access is requested to restricted systems to ensure there is strong identity verification for every access point. |
| Authorization | Authorization involves defining and enforcing what users are allowed to do and data they have access to once they have been authenticated. This policy mandates the usage of a role-based access control (RBAC) system, or something similar, to ensure the principle of least privilege is applied and users only have the minimum access rights required to effectively perform their job. This policy should apply when users attempt to access certain files or databases or when they perform operations. Applying this policy properly will ensure that access is appropriately limited based on the role of a user. |
| Accounting | Accounting incorporates tracking and logging the activities of users, including access to resources and actions that are performed. This policy requires the implementation of detailed logging of user logins, database changes, the addition of new users, changes to user access levels, and files accessed by users. The purpose of this policy is to maintain a paper trail for every action taken within the system, simultaneously providing accountability and supporting compliance requirements. This policy must be applied to all actions within the system to allow for a complete and detailed log that can be analyzed for suspicious behavior and ensure the system complies with regulatory standards. |

## 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 | 10/13/2024 | Final Policy | Jonathan Warner |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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