**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 | Input data from untrusted sources must be validated because inputs may be vectors for malicious code to be injected into the system. Bad inputs may exploit vulnerabilities such as buffer overflows and must be checked before running. |
| 1. Heed Compiler Warnings | Even though compiler warnings don’t prevent compilation by default, warnings still should be taken seriously because they can often be security related. Compiler warnings should be investigated, and the underlying issue corrected. |
| 1. Architect and Design for Security Policies | The system should be built with security principles and policies in mind from the start. If the system is designed securely from the beginning, instead of making security an afterthought, then maintaining the system in the future will be easier. |
| 1. Keep It Simple | Minimize complexity so that engineers can understand and maintain the system more easily. Move complexity can increase the attack surface of a system, giving it more points where it can be subverted. |
| 1. Default Deny | Users should be explicitly given privileges for certain actions or to access certain data. This keeps the list of permissioned users as small as it needs to be, so there are fewer attack vectors. |
| 1. Adhere to the Principle of Least Privilege | Like “Default Deny,” users should have little to no privileges to perform actions or access data on a system unless necessary. Users may be elevated to different privilege levels on an as-needed basis. |
| 1. Sanitize Data Sent to Other Systems | Only the data needed should be sent to other systems, while the rest may be trimmed off in order to reduce the attack surface and the size of the payload, improving security and bandwidth usage. |
| 1. Practice Defense in Depth | There should be many layers that make up the security system, so that if one layer is compromised, another layer can act as a backup. |
| 1. Use Effective Quality Assurance Techniques | QA techniques can include peer review of code, static analysis checkers, and unit tests on source code. Networks and firewalls should have security scans performed on them. |
| 1. Adopt a Secure Coding Standard | Engineers who check code into version control should all know the secure coding standards required of their code. There should be a mix of human and automated tests to determine whether code is secure and fit to be merged into the master branch. |

## 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 | Never qualify a reference type with const or volatile. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a const-qualified reference to a char is formed instead of a reference to a const-qualified char. This results in undefined behavior. |
| #include <iostream>    **void** f(**char** c) {  **char** &**const** p = c;  p = 'p'; // Error: read-only variable is not assignable  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the const qualifier. |
| #include <iostream>    **void** f(**char** c) {  **char** &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings – This standard is a general best practice for C++, the nonobservance of which will likely result in a compiler warning but will still compile. Ignoring this warning will result in undefined behavior. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL52 |  |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |
| Polyspace Bug Finder | R2020a | CERT C++: DCL52-CPP | Checks for:     * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| Clang | 3.9 |  | Clang checksMs an error without the need to specify any special flags or options. |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Copy operations must not mutate the source object. |

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

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

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings – This standard is a general best practice for C++, the nonobservance of which will likely result in a compiler warning but will still compile. Ignoring this warning will result in undefined behavior. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-OOP58-a | Copy operations must not mutate the source object |
| Polyspace Bug Finder | R2020a | CERT C++: OOP58-CPP | Checks for copy operation modifying source operand (rule partially covered) |
| PRQA QA-C++ | 4.4 | 4075 |  |
|  |  |  |  |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    **void** f() {  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

|  |
| --- |
| **Principles(s):** Validate Input Data – This principle applies to input being assigned to a string. Failing to follow this principle can lead to buffer overflows. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator    Buffer overrun  Type overrun |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++test | 2020.2 | 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 | R2020a | CERT C++: STR50-CPP | Checks for:     * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation   Rule partially covered. |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Sanitize data passed to complex subsystems. |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.    The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| **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 = '\_';  } |

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

|  |
| --- |
| **Principles(s):** Sanitize Data Sent to Other Systems – Concerning SQL injection, which is a common attack vector, commands to a database must be parsed and free of any malicious, arbitrary SQL commands.  Validate Input Data – All user input should be considered untrusted, and therefore must be confirmed to be free of ill-formed or malicious code before the data is evaluated. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0.p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |
| Polyspace Bug Finder | R2020a | CERT C: Rec. STR02-C | Checks for:    Execution of externally controlled command  Command executed from externally controlled path  Library loaded from externally controlled path  Rec. partially covered. |

### Coding Standard 5

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

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {  S \*s = **new** S;  // ...  **delete** s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| When possible, use automatic storage duration instead of dynamic storage duration. Since s is not required to live beyond the scope of g(), this compliant solution uses automatic storage duration to limit the lifetime of s to the scope of g(). |
| **struct** S {  **void** f();  };    **void** g() {  S s;  // ...  s.f();  } |

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

|  |
| --- |
| **Principles(s):** Keep it Simple – The standard library has constructs that make memory management in C++ easier and less reliant on the ability of the individual developer to manage memory manually. These include smart pointers and the “new” and “delete” keywords. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer. |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed. |
| Polyspace Bug Finder | R2020a | CERT C++: MEM50-CPP | Checks for:     * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assert liberally. |

| **Noncompliant Code** |
| --- |
| Not using assertions can make debugging a longer process. |
| **inline** Value \*getOperand(unsigned I) {  **return**Operands[I];} |

| **Compliant Code** |
| --- |
| Use the “assert” macro to its fullest. Check all your preconditions and assumptions, you never know when a bug (not necessarily even yours) might be caught early by an assertion, which reduces debugging time dramatically. The “<cassert>” header file is probably already included by the header files you are using, so it doesn’t cost anything to use it.    To further assist with debugging, make sure to put an error message in the assertion statement, which is printed if the assertion is tripped. This helps the poor debugger make sense of why an assertion is being made and enforced, and hopefully what to do about it. Here is one complete example: |
| **inline** Value \*getOperand(unsigned I) {  assert(I < Operands.size() && "getOperand() out of range!");  **return**Operands[I];  } |

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques – Assertions are one of many ways to continuously check that code is running correctly, and should be a part of a security strategy. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Coding Standard 7

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

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

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

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

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques – Unhandled exceptions can result in abnormal process termination where the stack can be left unwound, and memory may not be properly deallocated. Abnormal terminations can also result in printing memory addresses to the user, which can be used to create an exploit. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2020a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | STD-008-CPP | Do not read uninitialized memory. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    **void** f() {  **int** i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>    **void** f() {  **int** i = 0;  std::cout << i;  } |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings – Reading uninitialized memory will not result in a compiler error, but it will result in the object having an indeterminate value until replaced. This is an undefined behavior that is easy to avoid by initializing variables before they are used. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Polyspace Bug Finder | R2020a | CERT C++: EXP53-CPP | Checks for:     * Non-initialized variable * Non-initialized pointer   Rule partially covered. |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initialization | STD-009-CPP | Do not modify the standard namespaces. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the declaration of x is added to the namespace std, resulting in undefined behavior. |
| **namespace** std {  **int** x;  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes the intention of the programmer was to place the declaration of x into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name. |
| **namespace** nonstd {  **int** x;  } |

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

|  |
| --- |
| **Principles(s):** Keep it Simple – Modifying the standard namespaces adds unnecessary complication. If needed, custom namespaces without reserved names can be created, but it is often wise to evaluate whether it is really needed. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL58 |  |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces 'std' and 'posix' |
| Polyspace Bug Finder | R2020a | CERT C++: DCL58-CPP | Checks for modification of standard namespaces (rule fully covered) |
| PRQA QA-C++ | 4.4 | 4032, 4035, 4631 |  |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | STD-010-CPP | Prefer special member functions and overloaded operators to C Standard Library functions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a nontrivial class object is initialized by calling its default constructor but is later reinitialized to its default state using std::memset(), which does not properly reinitialize the object. Improper reinitialization leads to class invariants not holding in later uses of the object. |
| #include <cstring>  #include <iostream>    **class** C {  **int** scalingFactor;  **int** otherData;    **public**:  C() : scalingFactor(1) {}    **void** set\_other\_data(**int** i);  **int** f(**int** i) {  **return** i / scalingFactor;  }  // ...  };    **void** f() {  C c;    // ... Code that mutates c ...    // Reinitialize c to its default state  std::**memset**(&c, 0, **sizeof**(C));    std::cout << c.f(100) << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to std::memset() is replaced with a default-initialized copy-and-swap operation called clear(). This operation ensures that the object is initialized to its default state properly, and it behaves properly for object types that have optimized assignment operators that fail to clear all data members of the object being assigned into. |
| #include <iostream>  #include <utility>    **class** C {  **int** scalingFactor;  **int** otherData;    **public**:  C() : scalingFactor(1) {}    **void** set\_other\_data(**int** i);  **int** f(**int** i) {  **return** i / scalingFactor;  }  // ...  };    **template** <**typename** T>  T& clear(T &o) {  **using** std::swap;  T empty;  swap(o, empty);  **return** o;  }    **void** f() {  C c;    // ... Code that mutates c ...    // Reinitialize c to its default state  clear(c);    std::cout << c.f(100) << std::endl;  } |

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

|  |
| --- |
| **Principles(s):** Keep it Simple – Many classes and methods from the C standard library have been reimplemented into the C++ standard library with better awareness of security vulnerabilities. Using C standard library code in C++ is an obsolete practice. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | stdlib-use-ato  stdlib-use  stdlib-use-getenv  stdlib-use-system  include-time  stdlib-use-string-unbounded | Partially checked |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-OOP57-a  CERT\_CPP-OOP57-b | Do not initialize objects with a non-trivial class type using C standard library functions  Do not compare objects of nonstandard-layout class type with C standard library functions |
| Polyspace Bug Finder | R2020a | CERT C++: OOP57-CPP | Checks for bytewise operations on nontrivial class object (rule fully covered) |

## 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.

[Insert your written explanations here.]

## 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 | Low | Unlikely | Low | P3 | L3 |
| STD-002-CPP | Low | Likely | Low | P9 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Likely | Low | P3 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | Medium | P12 | L1 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-010-CPP | High | Probable | High | P6 | L2 |

## 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 refers to encrypted data that is being stored without being used by an individual or an application. This type of data is often targeted by hackers because it’s often in a centralized repository. Data should be encrypted at rest so that if a breach occurs, then the hacker will not be able to use the encrypted data, assuming they do not have the decryption key. |
| Encryption in flight | Encryption in flight refers to encrypted data that is being transmitted from a sender to one or more receivers. Encryption in flight can mitigate against man-in-the-middle attacks such as packet sniffing, where attackers intercept and read packets of data as they are sent across a network. One everyday example of encryption in flight is HTTPS. |
| Encryption in use | Encryption in use refers to encrypted data that is being generated, viewed, processed, or updated by one or more applications. Since the process of encrypting data is computationally expensive and can very easily slow down or crash the application accessing it, an alternative practice is to design and architect the application so that encryption in use is not necessary. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the concept of identifying a unique user by matching a username and password. Authentication is the first stage in a user management system, since a username is required to log in. |
| Authorization | Authorization refers to the permissions each user has to access certain data or services in the system. For example, a basic user may be allowed to edit resources they’ve created, while admin users can edit any resource. To follow the Principle of Least Privilege, authenticated users should have little to no permissions until explicitly granted. |
| Accounting | Accounting refers to tracking all user activities on the system, such as when and where a login occurred, which data are being accessed by the user, and services that they run on the system. This is important to provide an auditable paper trail, so that in the event of a data breach or improper use of the system, administrators can follow the logs and see who performed each action and at what time. |

**\***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 |  |
| 1.1 | 3/21/2021 | Draft 1 | Walter Augustine | Professor Olga Mill |
| 1.2 | 4/11/2021 | Final Draft | Walter Augustine | Professor Olga Mill |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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