**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 | Validate all data that comes from untrusted external sources. This can reduce many vulnerabilities and problems. Sources like interfaces and user controlled files should be checked before use. |
| 1. Heed Compiler Warnings | Compiler warnings should be fixed before software is considered safe. Warnings should be taken seriously. Static and dynamic tools should be used to check and fix software that may have bugs or vulnerabilities. |
| 1. Architect and Design for Security Policies | Software should be written with security being a major influence on the design. Systems should be split up with its own specific set of privilege and security. |
| 1. Keep It Simple | Software should be simple. Complex programs can lead to more errors and problems that can allow access to unauthorized users. Complex code is hard to debug and maintain. |
| 1. Default Deny | Nobody should have access by default. Everyone should be denied access as a safety precaution and individual users setup so that they can have access on a need to know type of system. |
| 1. Adhere to the Principle of Least Privilege | The least number of privileges necessary to complete a job should be used. The systems should be able to run with the least amount of access as possible. |
| 1. Sanitize Data Sent to Other Systems | All data that is to be passed onto a system should be checked and sanitized. This includes things like databases and off-the-shelf components. |
| 1. Practice Defense in Depth | Multiple layers of defense should be used to minimize risk. If an exploit get through one layer there is a chance that another could catch and stop the exploit before damage is done. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques like auditing and testing can be used to help ensure security. Outside testing and input can be good for catching problems before they start. |
| 1. Adopt a Secure Coding Standard | A good secure coding standard should be developed and used to keep systems secure. |

### 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** | [INT-30-C] | Ensure unsigned integer operations do not wrap. |

| **Noncompliant Code** |
| --- |
| Noncompliant code can result in an unsigned integer wrap during the addition of unsigned operands. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {    unsigned int usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Compliant solutions perform a precondition test of the operands of the addition to guarantee 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;    }    /\* ... \*/  } |

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

| **Principles(s):** [Head Compiler Warnings. Code vulnerabilities are often pointed out by compilers. They should be fixed first before any builds to prevent unwarranted behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.1p0 | **LANG.CAST.PC.AV LANG.CAST.PC.CONST2PTR LANG.CAST.PC.INT**  **LANG.CAST.COERCE LANG.CAST.VALUE**  **ALLOC.SIZE.TRUNC MISC.MEM.SIZE.TRUNC**  **LANG.MEM.TBA** | Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer  Coercion alters value Cast alters value  Truncation of allocation size Truncation of size  Tainted buffer access |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Can detect violations of this rule. However, false warnings may be raised if limits.h is included |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity)\* | 2017.07 | **NEGATIVE\_RETURNS**  **REVERSE\_NEGATIVE**  **MISRA\_CAST** | Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior  Can find instances where a negativity check occurs after the negative value has been used for something else  Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [DCL-52-CPP] | Never qualify a reference type with const or volatile |

| **Noncompliant Code** |
| --- |
| const-qualified reference to a char is formed instead of a reference to a const-qualified char |
| #include <iostream>    void f(char c) {    char &const p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| compliant solutions remove 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):** A const or volatile reference type may result in undefined behavior instead of a fatal diagnostic, causing unexpected values to be stored and leading to possible data integrity violations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL52** |  |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.3 | **C++0014** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2022.3 | **CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-DCL52-a** | Never qualify a reference type with 'const' or 'volatile' |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STR-53-CPP] | Range check element access. |

| **Noncompliant Code** |
| --- |
| the value returned by the call to get\_index() may be greater than the number of elements stored in the string, resulting in undefined behavior. |
| #include <string>    extern std::size\_t get\_index();    void f() {    std::string s("01234567");    s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the std::basic\_string::at() function, which behaves in a similar fashion to the index operator[] but throws a std::out\_of\_range exception if pos >= size(). |
| #include <stdexcept>  #include <string>  extern std::size\_t get\_index();    void f() {    std::string s("01234567");    try {      s.at(get\_index()) = '1';    } catch (std::out\_of\_range &) {      // Handle error    }  } |

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

| **Principles(s):** Unchecked element access can lead to out-of-bound reads and writes and write-anywhere exploits. These exploits can, in turn, lead to the execution of arbitrary code with the permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **assert\_failure** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.1p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.3 | **C++3162, C++3163, C++3164, C++3165** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-STR53-a** | Guarantee that container indices are within the valid range |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STR-02-C] | 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** |
| --- |
| 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):** failure to sanitize data passed to a complex subsystem can lead to an injection attack, data integrity issues, and a loss of sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.04 |  | Supported by stubbing/taint analysis |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.1p0 | **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 |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2022.3 | **NNTS.TAINTED** **SV.TAINTED.INJECTION** |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-50-CPP] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| 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** |
| --- |
| the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

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

| **Principles(s):** Reading previously dynamically allocated memory after it has been deallocated can lead to abnormal program termination and denial-of-service attacks. Writing memory that has been deallocated can lead to the execution of arbitrary code with the permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **dangling\_pointer\_use** |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MEM50** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/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. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.1p0 | **ALLOC.UAF** | Use after free |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [MSC-11-C] | Incorporate diagnostic tests using assertions |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char \*)malloc(len + 1);    assert(NULL != dup);      memcpy(dup, c\_str, len + 1);    return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char\*)malloc(len + 1);    /\* Detect and handle memory allocation error \*/    if (NULL == dup) {        return NULL;    }      memcpy(dup, c\_str, len + 1);    return dup;  } |

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

| **Principles(s):** Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The absence of assertions, however, does not mean that code is incorrect. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.1p0 | **LANG.FUNCS.ASSERTS** | Not enough assertions |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **ASSERT\_SIDE\_EFFECT** | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.1 | **CERT\_C-MSC11-a** | Assert liberally to document internal assumptions and invariants |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [ERR-55-CPP] | If a function throws an exception other than one allowed by its exception-specification, it can lead to an implementation-defined termination of the program  If a function declared with a dynamic-exception-specification throws an exception of a type that would not match the exception-specification, the function std::unexpected() is called. The behavior of this function can be overridden but, by default, causes an exception of std::bad\_exception to be thrown. Unless std::bad\_exception is listed in the exception-specification, the function std::terminate() will be called.  Similarly, if a function declared with a noexcept-specification throws an exception of a type that would cause the noexcept-specification to evaluate to false, the function std::terminate() will be called.  Calling std::terminate() leads to implementation-defined termination of the program. To prevent abnormal termination of the program, any function that declares an exception-specification should restrict itself, as well as any functions it calls, to throwing only allowed exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) noexcept(true) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) {    v.resize(s); // May throw, but that is okay  } |

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

| If a function throws an exception other than one allowed by its exception-specification, it can lead to an implementation-defined termination of the program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **unhandled-throw-noexcept** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR55** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.1p0 | **LANG.STRUCT.EXCP.THROW** | Use of throw |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.3 | **C++4035, C++4036, C++4632** |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory protection | [MEM-53-CPP] | Explicitly construct and destruct objects when manually managing object lifetime |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a class with nontrivial initialization (due to the presence of a user-provided constructor) is created with a call to std::malloc(). However, the constructor for the object is never called, resulting in undefined behavior when the class is later accessed by calling s->f(). |
| #include <cstdlib>    struct S {    S();      void f();  };    void g() {    S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));      s->f();      std::free(s);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructor and destructor are both explicitly called. Further, to reduce the possibility of the object being used outside of its lifetime, the underlying storage is a separate variable from the live object. |
| #include <cstdlib>  #include <new>    struct S {    S();      void f();  };    void g() {    void \*ptr = std::malloc(sizeof(S));    S \*s = new (ptr) S;      s->f();      s->~S();    std::free(ptr);  } |

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

| **Principles(s):** properly construct or destroy an object to prevent undefined behavior and accidental information exposure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.3 | **C++4761, C++4762, C++4766, C++4767** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-MEM53-a** | Do not invoke malloc/realloc for objects having constructors |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2022b | [CERT C++: MEM53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem53cpp.html) | Checks for objects allocated but not initialized (rule fully covered). |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.21 | [**V630**](https://pvs-studio.com/en/docs/warnings/v630/)**,**[**V749**](https://pvs-studio.com/en/docs/warnings/v749/) |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object oriented programming | [OOP-54-CPP] | Gracefully handle self-copy assignment. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the copy assignment operator does not protect against self-copy assignment. If self-copy assignment occurs, this->s1 is deleted, which results in rhs.s1 also being deleted. The invalidated memory for rhs.s1 is then passed into the copy constructor for S, which can result in dereferencing an [invalid pointer](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-invalidpointer). |
| #include <new>    struct S { S(const S &) noexcept; /\* ... \*/ };    class T {    int n;    S \*s1;    public:    T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(\*rhs.s1) : nullptr) {}    ~T() { delete s1; }      // ...      T& operator=(const T &rhs) {      n = rhs.n;      delete s1;      s1 = new S(\*rhs.s1);      return \*this;    }  }; |

| **Compliant Code** |
| --- |
| This compliant solution guards against self-copy assignment by testing whether the given parameter is the same as this. If self-copy assignment occurs, then operator= does nothing; otherwise, the copy proceeds as in the original example. |
| #include <new>    struct S { S(const S &) noexcept; /\* ... \*/ };    class T {    int n;    S \*s1;    public:    T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(\*rhs.s1) : nullptr) {}    ~T() { delete s1; }      // ...      T& operator=(const T &rhs) {      if (this != &rhs) {        n = rhs.n;        delete s1;        try {          s1 = new S(\*rhs.s1);        } catch (std::bad\_alloc &) {          s1 = nullptr; // For basic exception guarantees          throw;        }      }      return \*this;    }  }; |

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

| **Principles(s):** User-provided copy operators must properly handle self-copy assignment. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | **P2** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **dangling\_pointer\_use** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 9.0 (r361550) | cert-oop54-cpp | Checked by clang-tidy. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.1p0 | **IO.DC** **ALLOC.DF** **ALLOC.LEAK** **LANG.MEM.NPD** **LANG.STRUCT.RC** **IO.UAC** **ALLOC.UAF** | Double Close Double Free Leak Null Pointer Dereference Redundant Condition Use After Close Use After Free |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.3 | **C++4072, C++4073, C++4075, C++4076** |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | [CON-50-CPP] | Do not destroy a mutex while it is locked. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example creates several threads that each invoke the do\_work() function, passing a unique number as an ID.  Unfortunately, this code contains a race condition, allowing the mutex to be destroyed while it is still owned, because start\_threads() may invoke the mutex's destructor before all of the threads have exited. |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    void start\_threads() {    std::thread threads[maxThreads];    std::mutex m;      for (size\_t i = 0; i < maxThreads; ++i) {      threads[i] = std::thread(do\_work, i, &m);    }  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates the race condition by extending the lifetime of the mutex. |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    std::mutex m;    void start\_threads() {    std::thread threads[maxThreads];      for (size\_t i = 0; i < maxThreads; ++i) {      threads[i] = std::thread(do\_work, i, &m);    }  } |

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

| **Principles(s):** Mutex objects are used to protect shared data from being concurrently accessed. If a mutex object is destroyed while a thread is blocked waiting for the lock, critical sections and shared data are no longer protected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.1p0 | **CONCURRENCY.LOCALARG** | Local Variable Passed to Thread |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.3 | **C++4961, C++4962** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2022.3 | **CERT.CONC.MUTEX.DESTROY\_WHILE\_LOCKED** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-CON50-a** | Do not destroy another thread's mutex |

### 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 | High | Unlikely | Medium | High | 2 |
| INT-30-C | High | Probable | High | P6 | L2 |
| DCL-52-CPP | Low | Unlikely | Low | **P3** | **L3** |
| STR-53-CPP | High | Unlikely | Medium | **P6** | **L2** |
| STR-02-C | High | Likely | Medium | **P18** | **L1** |
| MEM-50-C | High | Likely | Medium | **P18** | **L1** |
| MSC-11-C | Low | Unlikely | High | **P1** | **L3** |
| ERR-55-CPP | Low | Likely | Low | **P9** | **L2** |
| MEM-53-CPP | High | Likely | Medium | **P18** | **L1** |
| OOP-54-CPP | Low | Probable | High | **P2** | **L3** |
| CON-50-CPP | Medium | Probable | High | **P4** | **L3** |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest protects data that is stored in formats like a database or in devices like phones and computers. There are options like AxCrypt for use. Encryption in rest is important to keep data from being stolen or accessed by unauthorized individuals. |
| Encryption at flight | Encryption in flight is the act of using secure channels to send data. This is done by the use of TSL or SFTP. This should always be used to help prevent unauthorized access and to prevent leaks. |
| Encryption in use | Encryption in use is the practice of encrypting memory even while it is use. This helps prevent crashes that result in memory dumps. Bitlocker is an example that encrypts and requires the use of a key. This practice should also be used when data is being accessed/used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies a user says they really are who they are and that they should be allowed access to information. This is done by the use of tools like passwords, certificates, etc. and helps prevent the loss or theft of data. |
| Authorization | Authorization works with authentication and verifies that a user should be allowed access to data/information/files. This prevents unauthorized users gaining access to privileged information. |
| Accounting | Accounting is the process of recording and keeping track of who access what information when and how. Information is stored like account numbers, usernames, passwords and dates and times of when a particular piece of information was accessed or used. If a breach or leak occurs, this provides information of where, when and who did it. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 12/2/2022 | Updated Version | Bob Johnson |  |
| [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 |