**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 data that is being input into the program. Verify external data to ensure that it is not corrupt or invalid based on the expected input data to prevent damage to the program or operating system. |
| 1. Heed Compiler Warnings | Pay careful attention to compiler warnings and fix them. Modifying the program code to satisfy warnings brought about during compilation will ensure that threat agents cannot get into the program. |
| 1. Architect and Design for Security Policies | Implement security policies throughout software architecture and design. Enforcing security policies during development ensures the production of secure software. |
| 1. Keep It Simple | Keep program code short and simple. Complex designs tend to have more errors and warnings than short and simple code. |
| 1. Default Deny | Deny access to everyone attempting to get into the system until they can verify authorization credentials. “To block all inbound and outbound traffic that has not been expressly permitted by firewall policy” (Deny by default - glossary). |
| 1. Adhere to the Principle of Least Privilege | Provide limited access to the system. Users should only have the least amount of privilege necessary to perform the job or task. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data that is passed through subsystems. Mitigates injection attacks by hackers attempting to manipulate subsystem components. |
| 1. Practice Defense in Depth | Practice DiD by using multiple, redundant layers of security. Having multiple, redundant layers helps ensure that systems are protected as efficiently as possible. If one layer fails, another layer of defense is there to catch and prevent exploitable vulnerabilities. |
| 1. Use Effective Quality Assurance Techniques | Using effective QA techniques increases likelihood of identifying and eliminating flaws and vulnerabilities. Security reviews and multiple testing phases ensure a more secure system. |
| 1. Adopt a Secure Coding Standard | Implement a secure coding standard. Every platform and language should have a set secure coding standard for development. |

### 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] | Do not cast an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| Checks whether a given value is within range of acceptable enumeration values. After casting the type, it  might not be able to represent the given integer value. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| The compliant solution checks the value represented by the enumeration type before performing the  conversion to guarantee the conversion doesn't result in an unspecified value. In turn is restricts the  converted value to one specific enumerator type. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |
| This compliant solution uses a scoped enumeration, which has a fixed underlying int type by default, allowing any value from the parameter to be converted into a valid enumeration value. It does not restrict the converted value to one for which there is a specific enumerator value, but it could do so as shown in the previous compliant solution. |
| enum class EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** It is possible for unspecified values to result in a buffer overflow, leading to the  execution of arbitrary code by an attacker. However, because enumerators are rarely used for  indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will  result in data integrity violations rather than arbitrary code execution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ | 2020.2 | CERT\_CPP - INT50-a | An expression with enum underlying type shall only have test values corresponding to the enumerators of the enumeration |
| Axivion  Bauhaus Suite | 6.9.0 | CertC++ - INT50 |  |
| Helix QAC | 2021.1 |  |  |
| PRQA QA- C++ | 4.4 | 3013 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Use valid references, pointers, and iterators to reference elements of a  container |

| **Noncompliant Code** |
| --- |
| This noncompliant code example copies input into a std::string, replacing semicolon (;) characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    void f(const std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();    for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |
| In this noncompliant code example, data is invalidated after the call to replace(), and so its use in g() is undefined behavior. |
| #include <iostream>  #include <string>    extern void g(const char \*);    void f(std::string &exampleString) {  const char \*data = exampleString.data();  // ...  exampleString.replace(0, 2, "bb");  // ...  g(data);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    void f(const std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();    for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |
| In this compliant solution, the pointer to exampleString's internal buffer is not generated until after the modification from replace() has completed. |
| #include <iostream>  #include <string>    extern void g(const char \*);    void f(std::string &exampleString) {    // ...    exampleString.replace(0, 2, "bb");    // ...    g(exampleString.data());  } |

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

| **Principles(s):** Using invalid references, pointers, or iterators to reference elements of a container results in  undefined behavior |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++  Test | 2020.2 | CERT\_CPP-CTR51-a | Do not modify container while iterating over it |
| Astree  PVS-Studio | 20.10  7.07 | Overflow\_unpon\_dereference  V783 |  |
| Helix QAC | 2021.1 |  |  |

#### Coding Standard 3

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

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv().  However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior  when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string  object is constructed. |
| #include <cstdlib>  #include <string>    void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In  some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack  2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not  possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++  test | 2020.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Astree | 20.10 | Assert\_failure |  |
| Helix QAC | 2021.1 |  |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not store already-owned pointer value in an unrelated smart pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying  pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages.  Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a  double-free vulnerability. |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::shared\_ptr objects are related to one another through copy construction.  When the local, automatic variable p2 is destroyed, the use count for the shared pointer value is  decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for  the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant  solution also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local  variable. |
| #include <memory>  void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):** Passing a pointer value to a deallocation function that was not previously obtained by the  matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++  Test | 2020.2 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| Astree | 20.10 | Dangling\_pointer\_use |  |
| Helix QAC | 2021.1 |  |  |
| PVS - Studio | 7.01 | V1006 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the local variable space is passed as the expression to the placement  new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined  behavior due to ::operator delete() attempting to free memory that was not returned by ::operator new(). |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    delete s1;  } |
| In this noncompliant code example, two allocations are attempted within the same try block, and if either fails, the catch handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for i1 may result in passing ::operator delete() a value (in i2) that was not previously returned by a call to ::operator new(), resulting in undefined behavior. |
| #include <new>    void f() {    int \*i1, \*i2;    try {      i1 = new int;      i2 = new int;    } catch (std::bad\_alloc &) {      delete i1;      delete i2;    }  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. This  is one of the few times when explicitly invoking a destructor is warranted. |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;    // ...    s1->~S();  } |
| This compliant solution initializes both pointer values to nullptr, which is a valid value to pass to ::operator delete(). |
| #include <new>    void f() {    int \*i1 = nullptr, \*i2 = nullptr;    try {      i1 = new int;      i2 = new int;    } catch (std::bad\_alloc &) {      delete i1;      delete i2;    }  } |

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

| **Principles(s):** Passing a pointer value to a deallocation function that was not previously obtained by the  matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Invalid\_dynamic\_memory\_alocation\_d  angling\_pointer\_use |  |
| CodeSonar | 6.0p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM | Free non-heap variable  Double free  Type mismatch |
| Parasoft C/C++  Test | 2020.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in  corresponding calls to new/malloc  and delete/free  Always provide empty brackets ([])  for delete when deallocating  arrays  Both copy constructor and copy  assignment operator should be  declared for classes with a  nontrivial destructor  Properly deallocate dynamically  allocated resources |
| Helix QAC | 2021.1 |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Avoid information leakage when passing a class object across a trust boundary |

| **Noncompliant Code** |
| --- |
| This noncompliant code example runs in kernel space and copies data from arg to user space. However, padding bits may be used within the object, for example, to ensure the proper alignment of class data members. These padding bits may contain sensitive information that may then be leaked when the data is copied to user space, regardless of how the data is copied. |
| #include <cstddef>    struct test {    int a;    char b;    int c;  };    // Safely copy bytes to user space  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{1, 2, 3};    copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |
| In this noncompliant code example, arg is value-initialized through direct initialization. Because test does not have a user-provided default constructor, the value-initialization is preceded by a zero-initialization that guarantees the padding bits are initialized to 0 before any further initialization occurs. It is akin to using std::memset() to initialize all of the bits in the object to 0. |
| #include <cstddef>    struct test {    int a;    char b;    int c;  };    // Safely copy bytes to user space  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{};      arg.a = 1;    arg.b = 2;    arg.c = 3;      copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |
| This code example runs in kernel space and copies data from arg to user space. However, the padding bits within the object instance may contain sensitive information that will then be leaked when the data is copied to user space. |
| #include <cstddef>    class base {  public:    virtual ~base() = default;  };    class test : public virtual base {    alignas(32) double h;    char i;    unsigned j : 80;  protected:    unsigned k;    unsigned l : 4;    unsigned short m : 3;  public:    char n;    double o;      test(double h, char i, unsigned j, unsigned k, unsigned l, unsigned short m,         char n, double o) :      h(h), i(i), j(j), k(k), l(l), m(m), n(n), o(o) {}      virtual void foo();  };    // Safely copy bytes to user space.  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{0.0, 1, 2, 3, 4, 5, 6, 7.0};    copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |

| **Compliant Code** |
| --- |
| This compliant solution serializes the structure data before copying it to an untrusted context. |
| #include <cstddef>  #include <cstring>    struct test {    int a;    char b;    int c;  };    // Safely copy bytes to user space.  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{1, 2, 3};    // May be larger than strictly needed.    unsigned char buf[sizeof(arg)];    std::size\_t offset = 0;      std::memcpy(buf + offset, &arg.a, sizeof(arg.a));    offset += sizeof(arg.a);    std::memcpy(buf + offset, &arg.b, sizeof(arg.b));    offset += sizeof(arg.b);    std::memcpy(buf + offset, &arg.c, sizeof(arg.c));    offset += sizeof(arg.c);      copy\_to\_user(usr\_buf, buf, offset /\* size of info copied \*/);  } |
| Padding bits can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the implementation and target memory architecture. The following solution is specific to the x86-32 architecture. |
| #include <cstddef>    struct test {    int a;    char b;    char padding\_1, padding\_2, padding\_3;    int c;      test(int a, char b, int c) : a(a), b(b),      padding\_1(0), padding\_2(0), padding\_3(0),      c(c) {}  };  // Ensure c is the next byte after the last padding byte.  static\_assert(offsetof(test, c) == offsetof(test, padding\_3) + 1,                "Object contains intermediate padding");  // Ensure there is no trailing padding.  static\_assert(sizeof(test) == offsetof(test, c) + sizeof(int),                "Object contains trailing padding");        // Safely copy bytes to user space.  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{1, 2, 3};    copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |
| Due to the complexity of the data structure, this compliant solution serializes the object data before copying it to an untrusted context instead of attempting to account for all of the padding bytes manually. |
| #include <cstddef>  #include <cstring>    class base {  public:    virtual ~base() = default;  };  class test : public virtual base {    alignas(32) double h;    char i;    unsigned j : 80;  protected:    unsigned k;    unsigned l : 4;    unsigned short m : 3;  public:    char n;    double o;      test(double h, char i, unsigned j, unsigned k, unsigned l, unsigned short m,         char n, double o) :      h(h), i(i), j(j), k(k), l(l), m(m), n(n), o(o) {}      virtual void foo();    bool serialize(unsigned char \*buffer, std::size\_t &size) {      if (size < sizeof(test)) {        return false;      }        std::size\_t offset = 0;      std::memcpy(buffer + offset, &h, sizeof(h));      offset += sizeof(h);      std::memcpy(buffer + offset, &i, sizeof(i));      offset += sizeof(i);      unsigned loc\_j = j; // Only sizeof(unsigned) bits are valid, so this is not narrowing.      std::memcpy(buffer + offset, &loc\_j, sizeof(loc\_j));      offset += sizeof(loc\_j);      std::memcpy(buffer + offset, &k, sizeof(k));      offset += sizeof(k);      unsigned char loc\_l = l & 0b1111;      std::memcpy(buffer + offset, &loc\_l, sizeof(loc\_l));      offset += sizeof(loc\_l);      unsigned short loc\_m = m & 0b111;      std::memcpy(buffer + offset, &loc\_m, sizeof(loc\_m));      offset += sizeof(loc\_m);      std::memcpy(buffer + offset, &n, sizeof(n));      offset += sizeof(n);      std::memcpy(buffer + offset, &o, sizeof(o));      offset += sizeof(o);        size -= offset;      return true;    }  };    // Safely copy bytes to user space.  extern int copy\_to\_user(void \*dest, void \*src, size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{0.0, 1, 2, 3, 4, 5, 6, 7.0};      // May be larger than strictly needed, will be updated by    // calling serialize() to the size of the buffer remaining.    std::size\_t size = sizeof(arg);    unsigned char buf[sizeof(arg)];    if (arg.serialize(buf, size)) {      copy\_to\_user(usr\_buf, buf, sizeof(test) - size);    } else {      // Handle error    }  } |

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

| **Principles(s):** Padding bits might inadvertently contain sensitive data such as pointers to kernel data structures or passwords. A pointer to such a structure could be passed to other functions, causing information leakage. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-DCL55-a | A pointer to a structure should not be passed to a function that can copy data to the user space |
| Helix QAC | 2022.1 | C++4941, C++4942, C++4943 |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL55 |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions thrown before main() begins executing |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the constructor for S may throw an exception that is not caught when globalS  is constructed during program startup. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |
| In this noncompliant code example, the thread entry point function thread\_start() does not catch exceptions thrown by throwing\_func(). If the initial thread function exits because an exception is thrown, std::terminate() is called. |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start() {    throwing\_func();  }    void f() {    std::thread t(thread\_start);    t.join();  } |

| **Compliant Code** |
| --- |
| This compliant solution makes globalS into a local variable with static storage duration, allowing any  exceptions thrown during object construction to be caught because the constructor for S will be executed  the first time the function globalS() is called rather than at program startup. This solution does require the  programmer to modify source code so that previous uses of globalS are replaced by a function call to  globalS(). |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |
| In this compliant solution, the thread\_start() handles all exceptions and does not rethrow, allowing the thread to terminate normally. |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start(void) {    try {      throwing\_func();    } catch (...) {      // Handle error    }  }    void f() {    std::thread t(thread\_start);    t.join();  } |

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

| **Principles(s):** Throwing an exception that cannot be caught results in abnormal program termination and can  lead to denial-of-service attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++  Test | 2020.2 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after startup and before  termination of the program |
| Clang | 3.9 | Cert-eer58-cpp | Checked by clang-tidy |
| Rule Checker | 20.10 | potentially-throwing-static-initialization | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented | [STD-008-CPP] | Do not invoke virtual functions from constructors or destructors |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class attempts to seize and release an object's resources  through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls  B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| struct B {  B() { seize(); }  virtual ~B() { release(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  B::seize();  // Get derived resources...  }  void release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors call a nonvirtual, private member function  (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing  and releasing its own resources. |
| class B {  void seize\_mine();  void release\_mine();  public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  B::seize();  // Get derived resources...  }  void release() override {  // Release derived resources...  B::release();  }  }; |

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

| **Principles(s):** Calls in virtual constructors or destructors will never go to a more derived class than the executing one. Virtual functions in constructors or destructors are not virtual. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | virtual-call-in-constructor  invalid\_function\_pointer | Fully Checked |
| Clang | 3.9 | clang-analyzer-  alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Parasoft C/C++  test | 2020.2 | CERT\_CPP-OOP50-a  CERT\_CPP-OOP50-b  CERT\_CPP-OOP50-c  CERT\_CPP-OOP50-d | Avoid calling virtual functions from constructors  Avoid calling virtual functions from  destructors  Do not invoke class's virtual  functions from any of its  constructors  Do not invoke class's virtual  functions from its destructor |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | [STD-009-CPP] | A signal handler must be a plain old function. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the signal handler is declared as a static function. However, since all signal handler functions must have C language linkage, and C++ is the default language linkage for functions in C++, calling the signal handler results in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |
| #include <csignal>    static void sig\_handler(int sig) {  // Implementation details elided.  }    void install\_signal\_handler() {  if (SIG\_ERR == std::signal(SIGTERM, sig\_handler)) {  // Handle error  }  } |
| In this noncompliant code example, a signal handler calls a function that allows exceptions, and it attempts to handle any exceptions thrown. Because exceptions are not part of the common subset of C and C++ features, this example results in [implementation-defined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-implementation-definedbehavior). However, it is unlikely that the implementation's behavior will be suitable. For instance, on a stack-based architecture where a signal is generated asynchronously (instead of as a result of a call to std:abort() or std::raise()), it is possible that the stack frame is not properly initialized, causing stack tracing to be unreliable and preventing the exception from being caught properly. |
| #include <csignal>    static void g() noexcept(false);    extern "C" void sig\_handler(int sig) {    try {      g();    } catch (...) {      // Handle error    }  }    void install\_signal\_handler() {    if (SIG\_ERR == std::signal(SIGTERM, sig\_handler)) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| This compliant solution defines sig\_handler() as having C language linkage. As a consequence of declaring the signal handler with C language linkage, the signal handler will have external linkage rather than internal linkage. |
| #include <csignal>    extern "C" void sig\_handler(int sig) {    // Implementation details elided.  }    void install\_signal\_handler() {    if (SIG\_ERR == std::signal(SIGTERM, sig\_handler)) {      // Handle error    }  } |
| There is no compliant solution whereby g() can be called from the signal handler because it allows exceptions. Even if g() were implemented such that it handled all exceptions and was marked noexcept(true), it would still be noncompliant to call g() from a signal handler because g() would still use a feature that is not a part of the common subset of C and C++ features allowed by a signal handler. Therefore, this compliant solution removes the call to g() from the signal handler and instead polls a variable of type volatile sig\_atomic\_t periodically; if the variable is set to 1 in the signal handler, then g() is called to respond to the signal. |
| #include <csignal>    volatile sig\_atomic\_t signal\_flag = 0;  static void g() noexcept(false);    extern "C" void sig\_handler(int sig) {    signal\_flag = 1;  }    void install\_signal\_handler() {    if (SIG\_ERR == std::signal(SIGTERM, sig\_handler)) {      // Handle error    }  }    // Called periodically to poll the signal flag.  void poll\_signal\_flag() {    if (signal\_flag == 1) {      signal\_flag = 0;      try {        g();      } catch(...) {        // Handle error      }    }  } |

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

| **Principles(s):** Failing to use a plain old function as a signal handler can result in implementation-defined behavior as well as undefined behavior. Given the number of features that exist in C++ that do not also exist in C, the consequences that arise from failure to comply with this rule can range from benign (harmless) behavior to abnormal program termination, or even arbitrary code execution. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-MSC54-a | Properly define signal handlers |
| Helix QAC | 2022.1 | C++2888 |  |
| Klocwork | 2022.1 | CERT.MSC.SIG\_HANDLER.POF |  |
| PRQA QA-C++ | 4.4 | 2888 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | [STD-010-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |
| In this compliant solution, the stream is implicitly closed through [RAII](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-RAII) before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    {      std::fstream file(fileName);      if (!file.is\_open()) {        // Handle error        return;      }    } // file is closed properly here when it is destroyed    std::terminate();  } |

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

| **Principles(s):** Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2021b | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |
| Parasoft Insure++ |  |  | Runtime detection |

### 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 | P4 | 3 |
| STD-002-CPP | High | Probable | High | P6 | 2 |
| STD-003-CPP | High | Likely | Medium | P18 | 1 |
| STD-004-CPP | High | Likely | Medium | P18 | 1 |
| STD-005-CPP | High | Likely | Medium | P18 | 1 |
| STD-006-CPP | Low | Unlikely | High | P1 | 3 |
| STD-007-CPP | Low | Likely | Low | P9 | 2 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | 3 |
| STD-009-CPP | High | Probable | High | P6 | 2 |
| STD-010-CPP | Medium | Unlikely | Medium | P4 | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is an encryption model that encrypts and decrypts at rest data or stored data. It is used by encrypting data that is on a disk. This policy protects stored data from attacks. |
| Encryption at flight | Encryption at flight is encryption of data before it is transmitted. It is used by encrypting the data before transmission then computer system endpoints are authenticated. Upon arrival, the data is decrypted and verified. This policy ensures that data is not manipulated or stolen during transmission. |
| Encryption in use | Encryption in use is the encryption of data or an application that is currently being used and lets you run your computation on it. It is used through hardware based trusted execution environments or software-based encryption which enable active use of encrypted data. Hashing is one example of this. This policy is ensuring that any data that needs to be used cannot be openly viewed by the user. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process “of identifying a user, typically by having the user enter a valid username and valid password before access is granted” (Contributor, 2010). User credentials such as username and password enable the user to access the system. Some authentication methods include two-factor or multi-tier authentication methods. This policy is for user logins and ensures that only the intended users gain access to the system. |
| Authorization | Authorization is the process that “determines whether the user has the authority to issue such commands” (Contributor, 2010). It limits what the user is permitted to do based on what is required to complete specific tasks and determines the user’s level of access. These tasks include making changes to the database, the addition of new users, and files that can be accessed by the user. This policy prevents users from gaining too much or unnecessary access to specific parts of the system. |
| Accounting | Accounting is the process of logging user activity and “measures the resources a user consumes during access” (Contributor, 2010). It is used for “authorization control, billing, trend analysis, resource utilization, and capacity planning activities” (Contributor, 2010). This policy helps to track user activity. |

**\***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 | 03/19/2022 | First Revision | Cassidy Yohn |  |
| 1.2 | 04/11/2022 | Final Revision | Cassidy Yohn |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

References

*Cybersecurity modules: Security injections: Cyber4all @towson*. Cybersecurity Modules Security InjectionsCyber4All Towson RSS. (n.d.). Retrieved March 12, 2022, from https://cisserv1.towson.edu/~cssecinj/modules/other-modules/database/sql-injection-introduction/

Schiela , R. (2020, May 29). *Confluence*. SEI CERT C++ Coding Standard - SEI CERT C++ Coding Standard - Confluence. Retrieved March 19, 2022, from https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Contributor, T. T. (2010, November 16). *What is authentication, Authorization, and accounting (AAA)? - definition from whatis.com*. SearchSecurity. Retrieved April 11, 2022, from https://www.techtarget.com/searchsecurity/definition/authentication-authorization-and-accounting