**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 input from command line arguments, network interfaces, and environmental variables. Validation can reduce most vulnerabilities in software. |
| 1. Heed Compiler Warnings | Compile code with the highest warning level available for your compiler. Fix and eliminate warnings by changing the necessary code. Static and dynamic analysis tools can help discover vulnerabilities in code which need also to be removed. |
| 1. Architect and Design for Security Policies | Create and design software architectures that enforce security policies. Security policies that enforce and have subsystems which provide security for each privilege set. |
| 1. Keep It Simple | Keep the design and system as simple and small as possible. Complexity increases configuration, increase of chance of vulnerabilities, quality assurance, and the amount of security layers. |
| 1. Default Deny | No trust by default or access is denied by default. Access is permitted based on permission rather than exclusion. |
| 1. Adhere to the Principle of Least Privilege | Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker must execute arbitrary code with elevated privileges |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers might be able to perform SQL injection or command attacks. The processing performing the calling understands the context and is responsible for sanitizing the data before invoking the subsystem to prevent injection attacks. |
| 1. Practice Defense in Depth | Use multiple layers of defense in case one layer is penetrated another layer of defense could help limit or prevent further security consequences. |
| 1. Use Effective Quality Assurance Techniques | Early and throughout coding quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Automated testing, penetration testing, source code audits, and independent security audits all help to reduce vulnerabilities in the code. |
| 1. Adopt a Secure Coding Standard | Develop and apply a secure coding standard for your target development language and platform. |

## 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 | DCL50-CPP | Do not define a C-style variadic function |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior. |
| #include <cstdarg>    int add(int first, int second, ...) {    int r = first + second;    va\_list va;    va\_start(va, second);    while (int v = va\_arg(va, int)) {      r += v;    }    va\_end(va);    return r;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior. |
| #include <type\_traits>    template <typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Arg s) { return f + s; }    template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Ts... rest) {    return f + add(rest...);  } |

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

|  |
| --- |
| **Principles(s):** Incorrectly using a variadic function can result in [abnormal program termination](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination), unintended information disclosure, or execution of arbitrary code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Function-ellipsis | Fully checked |
| Axivion Bauhaus | 7.2.0 | CertC++ -DCL50 |  |
| Clang | 3.9 | Cert-dcl50-cpp | Checked by clang-tidy |
| CodeSonar | 6.0p0 | LANG.STRUCT.ELLIPSIS | Ellipsis |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Value | EXP54-CPP | Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in undefined behavior. |
| struct S {    void mem\_fn();  };    void f() {    S \*s;    s->mem\_fn();  }  [Noncompliant code block; code should be indented using 12-point Courier New font.] |

| **Compliant Code** |
| --- |
| In this compliant solution, storage is obtained for the pointer prior to calling S::mem\_fn(). |
| struct S {    void mem\_fn();  };    void f() {    S \*s = new S;    s->mem\_fn();    delete s;  } |

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

|  |
| --- |
| **Principles(s):** Referencing an object outside of its lifetime can result in an attacker being able to run arbitrary code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Return-reference-local dangling\_pointer\_use | Partially checked |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<> |
| CodeSonar | 6.0p0 | IO.UAC ALLOC.UAF | Use after close  Use after free |
| Helix QAU | 2021.1 | **C++2812, C++2813, C++2814, C++2930, C++2931, C++2932, C++2933, C++2934, C++4003, C++4026** | [Insert text.] |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| String Correctness | STR52-CPP | Use valid references, pointers, and iterators to references elements of a basic\_string |

| **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) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **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 : ' ');    }  } |

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

|  |
| --- |
| **Principles(s):** Using an invalid reference, pointer, or iterator to a string object could allow an attacker to run arbitrary code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.1 | C++4746, C++4747, C++4748, C++4748 |  |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-STR52-A | Use valid references, pointers, and iterators to reference elements of a basic\_string |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| SQL Injection | STR02-C | Sanitizew data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. |
| 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. |
| 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](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-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 | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported by stubbing/taint analysis |
| CodeSonar | 6.0p0 | **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 |
| [Insert text.] | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [Insert text.] | 2021.1 | [NNTS.TAINTED](https://support.roguewave.com/documentation/klocwork/en/current/certcandcsecurecodingstandardidsmappedtoklocworkcandccheckers/) [SV.TAINTED.INJECTION](https://support.roguewave.com/documentation/klocwork/en/current/certcandcsecurecodingstandardidsmappedtoklocworkcandccheckers/) |  |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | MEM00-C | Allocate and free memory in the same module, at the same level of abstraction |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows a double-free vulnerability resulting from memory being allocated and freed at differing levels of abstraction. In this example, memory for the list array is allocated in the process\_list() function. The array is then passed to the verify\_size() function that performs error checking on the size of the list. If the size of the list is below a minimum size, the memory allocated to the list is freed, and the function returns to the caller. The calling function then frees this same memory again, resulting in a double-free and potentially exploitable vulnerability. |
| enum { MIN\_SIZE\_ALLOWED = 32 };    int verify\_size(char \*list, size\_t size) {    if (size < MIN\_SIZE\_ALLOWED) {      /\* Handle error condition \*/      free(list);      return -1;    }    return 0;  }    void process\_list(size\_t number) {    char \*list = (char \*)malloc(number);    if (list == NULL) {      /\* Handle allocation error \*/    }      if (verify\_size(list, number) == -1) {        free(list);        return;    }      /\* Continue processing list \*/      free(list);  } |

| **Compliant Code** |
| --- |
| To correct this problem, the error-handling code in verify\_size() is modified so that it no longer frees list. This change ensures that list is freed only once, at the same level of abstraction, in the process\_list() function. |
| enum { MIN\_SIZE\_ALLOWED = 32 };    int verify\_size(const char \*list, size\_t size) {    if (size < MIN\_SIZE\_ALLOWED) {      /\* Handle error condition \*/      return -1;    }    return 0;  }    void process\_list(size\_t number) {    char \*list = (char \*)malloc(number);      if (list == NULL) {      /\* Handle allocation error \*/    }      if (verify\_size(list, number) == -1) {        free(list);        return;    }      /\* Continue processing list \*/      free(list);  } |

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

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PC-lint Plus | 1.4 | 449, 2434 | Partially supported |
| CodeSonar | 6.0p0 | **ALLOC.DF** **ALLOC.LEAK** | Double free Leak |
| Coverity | 6.5 | **RESOURCE\_LEAK** | Fully implemented |
| LDRA tool suite | 9.7.1 | **50 D** | Partially implemented |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Assertions | MSC11-C | Incorporate diagnostic tests using asssertions |

| **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. See also MEM11-C. Do not assume infinite heap space and void MEM32-C. Detect and handle memory allocation errors. |
| 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](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability). The absence of assertions, however, does not mean that code is incorrect. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **LANG.FUNCS.ASSERTS** | Not enough assertions |
| 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 | 2021.1 | **CERT\_C-MSC11-a** | Assert liberally to document internal assumptions and invariants |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exceptions | ERR51-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):** Allowing the application to [abnormally terminate](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination) can lead to resources not being freed, closed, and so on. It is frequently a vector for [denial-of-service attacks](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-denial-of-service). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **main-function-catch-all** **early-catch-all** | Partially checked |
| Axivion Bauhaus | 7.2.0 | **CertC++-ERR51** |  |
| Helix QAC | 2021.1 | **C++4035, C++4036, C++4037** |  |
| LDRA tool suite | 9.7.1 | **527 S** | Partially implemented |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | FIO51-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();  } |

**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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **ALLOC.LEAK** | Leak |
| Helix QAC | 2021.1 | **C++4786, C++4787, C++4788** |  |
| Klocwork | 2021.1 | [RH.LEAK](https://support.roguewave.com/documentation/klocwork/en/current/certcandcsecurecodingstandardidsmappedtoklocworkcandccheckers/) |  |
| Parasoft C/C++ test | 2021.1 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming (OOP) | OOP50-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() { seize\_mine(); }    virtual void release() { release\_mine(); }  };    class D : public B {    void seize\_mine();    void release\_mine();    public:    D() { seize\_mine(); }    virtual ~D() { release\_mine(); }    protected:    void seize() override {      B::seize();      seize\_mine();    }      void release() override {      release\_mine();      B::release();    }  }; |

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

|  |
| --- |
| **Principles(s):** Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **virtual-call-in-constructor** **invalid\_function\_pointer** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-OOP50** |  |
| Clang | 3.9 | clang-analyzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Helix QAC | 2021.1 | **C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282** |  |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency (CON) | CON53-CPP | Avoid deadlock by locking in a predefined order |

| **Noncompliant Code** |
| --- |
| The behavior of this noncompliant code example depends on the runtime environment and the platform's scheduler. The program is susceptible to deadlock if thread thr1 attempts to lock ba2's mutex at the same time thread thr2 attempts to lock ba1's mutex in the deposit() function. |
| #include <mutex>  #include <thread>    class BankAccount {    int balance;  public:    std::mutex balanceMutex;    BankAccount() = delete;    explicit BankAccount(int initialAmount) : balance(initialAmount) {}    int get\_balance() const { return balance; }    void set\_balance(int amount) { balance = amount; }  };    int deposit(BankAccount \*from, BankAccount \*to, int amount) {    std::lock\_guard<std::mutex> from\_lock(from->balanceMutex);      // Not enough balance to transfer.    if (from->get\_balance() < amount) {      return -1; // Indicate error    }    std::lock\_guard<std::mutex> to\_lock(to->balanceMutex);      from->set\_balance(from->get\_balance() - amount);    to->set\_balance(to->get\_balance() + amount);      return 0;  }    void f(BankAccount \*ba1, BankAccount \*ba2) {    // Perform the deposits.    std::thread thr1(deposit, ba1, ba2, 100);    std::thread thr2(deposit, ba2, ba1, 100);    thr1.join();    thr2.join();  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates the circular wait condition by establishing a predefined order for locking in the deposit() function. Each thread will lock on the basis of the BankAccount ID, which is set when the BankAccount object is initialized. |
| #include <atomic>  #include <mutex>  #include <thread>    class BankAccount {    static std::atomic<unsigned int> globalId;    const unsigned int id;    int balance;  public:    std::mutex balanceMutex;    BankAccount() = delete;    explicit BankAccount(int initialAmount) : id(globalId++), balance(initialAmount) {}    unsigned int get\_id() const { return id; }    int get\_balance() const { return balance; }    void set\_balance(int amount) { balance = amount; }  };    std::atomic<unsigned int> BankAccount::globalId(1);    int deposit(BankAccount \*from, BankAccount \*to, int amount) {    std::mutex \*first;    std::mutex \*second;      if (from->get\_id() == to->get\_id()) {      return -1; // Indicate error    }      // Ensure proper ordering for locking.    if (from->get\_id() < to->get\_id()) {      first = &from->balanceMutex;      second = &to->balanceMutex;    } else {      first = &to->balanceMutex;      second = &from->balanceMutex;    }    std::lock\_guard<std::mutex> firstLock(\*first);    std::lock\_guard<std::mutex> secondLock(\*second);      // Check for enough balance to transfer.    if (from->get\_balance() >= amount) {      from->set\_balance(from->get\_balance() - amount);      to->set\_balance(to->get\_balance() + amount);      return 0;    }    return -1;  }    void f(BankAccount \*ba1, BankAccount \*ba2) {    // Perform the deposits.    std::thread thr1(deposit, ba1, ba2, 100);    std::thread thr2(deposit, ba2, ba1, 100);    thr1.join();    thr2.join();  } |

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

|  |
| --- |
| **Principles(s):** Deadlock prevents multiple threads from progressing, halting program execution. A [denial-of-service attack](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-denial-of-service) is possible if the attacker can create the conditions for deadlock. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **CONCURRENCY.LOCK.ORDER** | Conflicting lock order |
| Coverity | 6.5 | **DEADLOCK** | Fully implemented |
| Helix QAC | 2021.1 | **C++1772, C++1773** |  |
| Parasoft C/C++ test | 2021.1 | **CERT\_CPP-CON53-a** | Do not acquire locks in different order |

## 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.

Within Build static analysis and testing can be implemented and automated. The static analysis and testing may find vulnerabilities and bugs before the Verify and test phase.

## 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 |
| [Insert text.] | High | Probable | High | High | 1 |
| [Insert text.] | High | Probable | High | High | 2 |
| [Insert text.] | High | Probable | Medium | High | 2 |
| [Insert text.] | High | Unlikely | Medium | High | 1 |
| [Insert text.] | Low | Probable | High | High | 1 |
| [Insert text.] | Low | Unlikely | Medium | Medium | 3 |
| [Insert text.] | Medium | Probable | Medium | Medium | 3 |
| [Insert text.] | Low | Unlikely | Medium | Medium | 3 |
| [Insert text.] | Low | Unlikely | Medium | Medium | 3 |
| [Insert text.] | Low | Probable | Medium | Medium | 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 at rest is designed to prevent the attacker from accessing the unencrypted data by ensuring the data is encrypted when on disk. If an attacker obtains a hard drive with encrypted data but not the encryption keys, the attacker must defeat the encryption to read the data. |
| Encryption at flight | The process of encrypting data while the data is being transmitted. In some applications, such as remote replication, data may be unencrypted while it is at rest on drive arrays, but encrypted while it is being transmitted to provide protection. |
| Encryption in use | In-Use encryption takes a new approach that ensures that sensitive data is never left unsecured, regardless of or lifecycle stage (at rest, in transit, or in use) source, or location (on premise, cloud, or hybrid). |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The method of identifying a user, typically by having the user enter a valid username and password before access to the network is granted. Authentication is based on each user having a unique set of login credentials for gaining network access. |
| Authorization | User must gain authorization for doing certain tasks. After logging in to a system, for instance, the user may try to issue commands. The authorization process determines whether the user has the authority to issue such commands. |
| Accounting | Accounting is carried out by logging session statistics and usage information. It is used for authorization control, billing, trend analysis, resource utilization, and planning for the data capacity required for business operations. |

**\***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 | 06/2021 | Add Policy Rules | Glenn Bacon |  |
| [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 |