**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Receiving inputs from untrusted sources require input validation to prevent software  vulnerabilities. External data sources, command line arguments, files, and network  interfaces are examples of untrusted sources that must be vetted before allowing them  to submit their input. This can prevent exploits of gaining unauthorized information or  access. |
| 1. Heed Compiler Warnings | Eliminate warnings by modifying the code. This prevents some of these warnings, no matter how small, from becoming larger issues with exploitive security flaws or bugs. Use static and dynamic analysis tools to detect additional flaws and correct them. |
| 1. Architect and Design for Security Policies | Create software architecture and design your software to enforce security policies to have a secure software. Having different levels of your software broken up into subsystems to create a more secure system where privileges can be assigned to access different parts of the system as needed. |
| 1. Keep It Simple | KISS, keeping the design simple allows easier user interaction and acceptance when reviewing code or adding to it. When you create a complex design, it increases the chances errors will generate and make configuration and use more difficult. |
| 1. Default Deny | Similar to principle of least privilege, by default your software should deny access and should only grant access under the right identity conditions. Having this as default keeps it secure from accidental access to higher security functions or access to data that is unauthorized. |
| 1. Adhere to the Principle of Least Privilege | Every process should execute with the principle of least privilege necessary for a given task. Elevated permissions should only be accessed for the least amount of time required to complete the privileged task. This reduces the opportunities an attacker has to execute arbitrary code with the higher privileges. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data passed to complex subsystems. This refers to command shells, relational databases, and commercial off-the-shelf (COTS) components. This is to prevent attackers from invoking unused functionality in those components through SQL commands or other injection attacks. The other complex subsystems being invoked do not understand the context of the call being made. |
| 1. Practice Defense in Depth | Manage risk with multiple defensive strategies, so that if one layer of defense turns inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability. Think of how a computer is protected, it has an antivirus that protects your computer from malware. In the event a virus or malware gets on your computer, a firewall on your network would block this virus from sending data out to IP points outside your network and blocking unwanted return traffic. |
| 1. Use Effective Quality Assurance Techniques | Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. It is also important to utilize outside security reviews to get a different perspective in potential vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for your target development language or 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** | DCL52-CPP | Never qualify a reference type with const or volatile. |

| **Noncompliant Code** |
| --- |
| In this example, a const qualified reference to a char is formed instead of a reference to a const qualified char. This results in an undefined behavior. |
| #include <iostream>    void f(char c) {    char &const p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| Removing the const qualified removes the undefined behavior. |
| #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):** Validate input data, the undefined behavior from a const reference to a char needs to be validated to obey the range requirements of the type. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-DCL52** |  |
| Helix QAC | 2021.2 | **C++0014** |  |
| Klocwork | 2021.4 | **CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE** |  |
| Polyspace Bug Finder | R2021b | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-DCL52-a** | Never qualify a reference type with 'const' or 'volatile' |
| PRQA QA-C++ | 4.4 | **0014** |  |
| Clang | 3.9 |  | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options. |
| SonarQube C/C++ Plugin | 4.10 | **S3708** |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | DCL53-CPP | Do not write syntactically ambiguous declarations. |

| **Noncompliant Code** |
| --- |
| An anonymous local variable of type std::unique\_lock is expected to lock and unlock the mutex m by virtue of RAII. But the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling a single-argument converting constructor or interpreted as declaring an object named m and default constructing it. The syntax used in this example defines the latter instead of the former and so the mutex object is never locked. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {    std::unique\_lock<std::mutex>(m);    shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lock object is given an identifier (other than m) and the proper converting constructor is called. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {    std::unique\_lock<std::mutex> lock(m);    shared\_resource += 42;  } |

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

| **Principles(s):** Validate input data, the declaration that is expected to lock and unlock mutex m is vague and can be interpreted as a different object being declared named m and thus default constructing it. If the data was not validated, it could cause unintentional results causing the mutex object to never lock. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | **C++2502, C++2510** |  |
| Klocwork | 2021.4 | **CERT.DCL.AMBIGUOUS\_DECL** |  |
| LDRA tool suite | 9.7.1 | **296 S** | Partially implemented |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-DCL53-a** **CERT\_CPP-DCL53-b CERT\_CPP-DCL53-c** | Parameter names in function declarations should not be enclosed in parentheses Local variable names in variable declarations should not be enclosed in parentheses Avoid function declarations that are syntactically ambiguous |
| Polyspace Bug Finder | R2021b | CERT C++: DCL53-CPP | Checks for declarations that can be confused between:   * Function and object declaration * Unnamed object or function parameter declaration   Rule fully covered. |
| PRQA QA-C++ | 4.4 | **2502, 2510** |  |
| Clang | 3.9 | -Wvexing-parse |  |
| SonarQube C/C++ Plugin | 4.10 | **S3468** |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR52-CPP | Use valid references, pointers, and iterators to reference 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) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| The value of the interator 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. ( std::string::insert() ) |
| #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):** Validate input data, Sanitize Data Sent to Other Systems and Keep it simple, The iterator loc becomes undefined after the first insert(). Information sent and received from other systems need to be properly sanitized and closed off to prevent unwanted interactions through exploits that could cause a system to be overloaded. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | **ALLOC.UAF** | Use After Free |
| Helix QAC | 2021.2 | **C++4746, C++4747, C++4748, C++4749** |  |
| Parasoft C/C++test | 2021.2 | **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 | Sanitize data passed to complex subsystems. |

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

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data. |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems and validate input, a data being passed to a buffer that uses string arguments to call another system without properly sanitizing and validating input could cause the system vulnerability of divulging too much information unless it can be properly validated and sanitized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 |  | Supported by stubbing/taint analysis |
| CodeSonar | 6.2p0 | **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 | 6.5 | **TAINTED\_STRING** | Fully implemented |
| Klocwork | 2021.4 | **NNTS.TAINTED** **SV.TAINTED.INJECTION** |  |
| LDRA tool suite | 9.7.1 | **108 D, 109 D** | Partially implemented |
| Parasoft C/C++test | 2021.2 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |
| Polyspace Bug Finder | R2021a | CERT C: Rec. STR02-C | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM52-CPP | Detect and handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| An array of int is created using **::operator new[] (std::size\_t)** and the results of the allocation are not checked. The function is marked as **noexcept**, so the caller assumes this function does not throw any exceptions. Because **::operator new[] (std::size\_t)** can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {    int \*copy = new int[size];    std::memcpy(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

| **Compliant Code** |
| --- |
| When using **std::nothrow** the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not **nullptr** |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {    int \*copy = new (std::nothrow) int[size];    if (!copy) {      // Handle error      return;    }    std::memcpy(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

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

| **Principles(s):** Heed compiler warnings, bypassing the throw of the array operator should not be handled without properly using pointers to the allocated space since a program should be handling errors properly and not causing conflicting issues in a program that leads to abnormal termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  |  |
| Coverity | 7.5 | **CHECKED\_RETURN** | Finds inconsistencies in how function call return values are handled |
| Helix QAC | 2021.2 | **C++3225, C++3226, C++3227, C++3228, C++3229, C++4632** |  |
| Klocwork | 2021.4 | **NPD.CHECK.CALL.MIGHT** **NPD.CHECK.CALL.MUST** **NPD.CHECK.MIGHT**  **NPD.CHECK.MUST**  **NPD.CONST.CALL** **NPD.CONST.DEREF** **NPD.FUNC.CALL.MIGHT** **NPD.FUNC.CALL.MUST** **NPD.FUNC.MIGHT** **NPD.FUNC.MUST**  **NPD.GEN.CALL.MIGHT NPD.GEN.CALL.MUST NPD.GEN.MIGHT NPD.GEN.MUST**  **RNPD.CALL RNPD.DEREF** |  |
| LDRA tool suite | 9.7.1 | **45 D** | Partially implemented |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-MEM52-a** **CERT\_CPP-MEM52-b** | Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2021b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| PRQA QA-C++ | 4.4 | **3225, 3226, 3227, 3228, 3229, 4632** |  |
| PVS-Studio | 7.16 | **V522, V668** |  |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| 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 using **malloc()** |
| 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):** Architect and Design for Security Policies, Heed Compiler warnings, and input validation, having a design that handles how a system security policy is implemented to prevent unwanted access to resources and data that could compromise a system by a denial of service attack. Validating inputs to prevent assert from overloading the memory by means of handling the memory allocation error with a pointer to null will prevent too many memory allocation processes happening at once that would cause an abrupt termination of the process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | **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 | 2021.2 | **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** |
| --- |
| 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** |
| --- |
| 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):** Heed compiler warnings, having to handle all exceptions and errors properly will help eliminate software vulnerabilities by properly handling all exceptions thrown. It will also be important to not have too much info given during an error that could help attackers from figuring out how the system works. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **main-function-catch-all early-catch-all** | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-ERR51** |  |
| Helix QAC | 2021.2 | **C++4035, C++4036, C++4037** |  |
| Klocwork | 2021.4 | **MISRA.CATCH.ALL** |  |
| LDRA tool suite | 9.7.1 | **527 S** | Partially implemented |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2021b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| PRQA QA-C++ | 4.4 | **4035, 4036, 4037** |  |
| RuleChecker | 20.10 | **main-function-catch-all early-catch-all** | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Management | MEM53-CPP | Explicitly construct and destruct objects when manually managing object lifetime |

| **Noncompliant Code** |
| --- |
| 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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior) 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** |
| --- |
| 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):** Sanitize data sent to other systems, keep it simple, You first want to keep it simple when designing objects that need to be constructed and destructed. By sanitizing the unwanted objects by destructing them before they are used outside their lifetime and causing storage issues. This could have led to memory overload which could crash a system or leak information by unwanted behaviors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | **C++4761, C++4762, C++4766, C++4767** |  |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-MEM53-a** | Do not invoke malloc/realloc for objects having constructors |
| PVS-Studio | 7.16 | **V630, V749** |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Type | DCL55-CPP | Avoid information leakage when passing a class object across a trust boundary. |

| **Noncompliant Code** |
| --- |
| This 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));  } |

| **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 \*/);  } |

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

| **Principles(s):** Input validation and sanitize data sent to other systems, must validate the input before it is sent to another system to prevent data being leaked when copied. This could lead to an memory management issue or buffer overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-DCL55** |  |
| Helix QAC | 2021.2 | **C++4941, C++4942, C++4943** |  |
| 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 |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Value | INT32-C | Ensure that operations on signed integers do not result in overflow. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in a signed integer overflow during the addition of the signed operands **si\_a** and **si\_b:** |
| void func(signed int si\_a, signed int si\_b) {    signed int sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution ensures that the addition operation cannot overflow, regardless of representation |
| #include <limits.h>    void f(signed int si\_a, signed int si\_b) {    signed int sum;    if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||        ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {      /\* Handle error \*/    } else {      sum = si\_a + si\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** Input validation, validating the input to prevent integer overflow by properly checking the values before letting them be passed through the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **integer-overflow** | Fully checked |
| CodeSonar | 6.2p0 | **ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Coverity | 2017.07 | **TAINTED\_SCALAR**  **BAD\_SHIFT** | Implemented |
| Helix QAC | 2021.3 | **C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863**  **C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863** |  |
| Klocwork | 2021.4 | **CWARN.NOEFFECT.OUTOFRANGE**  **NUM.OVERFLOW** |  |
| LDRA tool suite | 9.7.1 | **493 S, 494 S** | Partially implemented |
| Parasoft C/C++test | 2021.2 | **CERT\_C-INT32-a** **CERT\_C-INT32-b** **CERT\_C-INT32-c** | Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |
| Parasoft Insure++ |  |  | Runtime analysis |
| Polyspace Bug Finder | R2021a | CERT C: Rule INT32-C | Checks for:   * Integer overflow * Tainted division operand * Tainted modulo operand   Rule partially covered. |
| PRQA QA-C | 9.7 | **2800, 2801, 2802, 2803,**  **2860, 2861, 2862, 2863** | Fully implemented |
| PRQA QA-C++ | 4.4 | **2800, 2801, 2802, 2803,**  **2860, 2861, 2862, 2863** |  |
| PVS-Studio | 7.16 | **V1026, V1070, V5010** |  |
| TrustInSoft Analyzer | 1.38 | **signed\_overflow** | Exhaustively |

### 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.

When it comes to DevSecOps, automation is a critical component of ensuring rapid successful deployment of the standards defined by development, security and operations. Automation improves development by assisting software engineers with the detection of security threats in a quicker succession than a human user manually searching for them. With this time being saved, the automation gives better feedback quickly to start developing fixes that are designed, built, tested, and deployed. Security in automation allows the system to rapidly detect incidents as they happen and can handle them automatically by using pre-set action plans to given scenarios. For instance, a denial service causing memory load issues could have the system shut down access from the outside or take a system offline temporarily to flush the memory errors being generated. This would reduce the need for human response to manually addressing security issues. Operations in automation allow better consistency and accuracy by monitoring and detecting logs, analytics and event alerting for intrusion.

### 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 |
| DCL52-CPP | Low | Low | Unlikely | P3 | L3 |
| DCL53-CPP | Low | Unlikely | Medium | P2 | L3 |
| STR52-CPP | High | Probable | High | P6 | L2 |
| STR02-C | High | Likely | Medium | P18 | L1 |
| MEM52-CPP | High | Likely | Medium | P18 | L1 |
| MSC11-C | Low | Unlikely | High | P1 | L3 |
| ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| MEM53-CPP | High | Likely | Medium | P18 | L1 |
| DCL55-CPP | Low | Unlikely | High | P1 | L3 |
| INT32-C | High | Likely | High | P9 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest is the design process of preventing attackers from accessing unencrypted data by making sure all data on a system or disk is encrypted. This protects sensitive data with the encryption that unauthorized users cannot decrypt. An example of this could be patient data by HIPPA laws or a bank’s customer account numbers/records. |
| Encryption at flight | Encryption in flight is the process of encrypting data being transmitted between systems. This gives your system integrity when data is being moved from one place before it is decrypted on the other side for authorized users and prevents unauthorized users from intercepting data. |
| Encryption in use | Encryption in use is a similar to the principal of least privilege by restricting data by a user’s authentication credentials and keeping it secured to others without the proper credentials. This would apply to users that may have access to see bank account numbers or patient data while another user credentials, like an accountant not being able to see exactly what the patient was diagnosed with. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of a system comparing a user’s credentials with a database confirming this user’s credentials match and is permitted access to certain features or systems. This would be for users who need to access sensitive systems that contain important privileges or information that not all users should have access to. This would deny a user access if their credentials did not authenticate and prevent access to the system. |
| Authorization | After a user is authenticated, a user is then given authorization to a system up to X amount of privileges, which would be based on their role. This prevents resources from being over permitted which could cause security concerns and only gives the amount of resources a user’s role needs to be able to perform their duty/tasks. An example of this would be a network administrator’s ability to grant and deny access to users and a normal user not having this ability since they do not need access to alter the system. |
| Accounting | Accounting is like accountability for a user for which a system monitors the resources used by the user during their time in the system. This is like a log sheet that could track what a user does while logged in and when they do it. This is beneficial for tracking vulnerabilities or security breaches as a user could be tracked by their actions. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [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 |