**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | ‘Input Validation’ is the key term here, which is eliminating software vulnerabilities by verifying user or external database inputs into a program. |
| 1. Heed Compiler Warnings | This entails running the compiler alongside warning levels and checking those for vulnerabilities. Modify the code as the warning dictates to avoid excess security flaws. |
| 1. Architect and Design for Security Policies | When composing the design of software, security concerns should be considered and implemented within the design. This could result in changes to the way the software operates, but overall creates defensive layers. |
| 1. Keep It Simple | Essentially keep the code and design as simple and compact as possible. The more complex something gets, the more likely there will exist flaws or gaps in the product that can be exposed. |
| 1. Default Deny | One should base their accessibility decisions on permission levels rather than exclusionary methods. This ‘straightens’ out the security flow and denies those who don’t have specific permission. |
| 1. Adhere to the Principle of Least Privilege | A minimal number of elevated privileges should be employed to deny an attacker excess opportunity to exploit software using said elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | Subsystems such as command shell, relational databases, and COTS components should have data passed to them first sanitized to avoid attackers using injection attacks when unanticipated. |
| 1. Practice Defense in Depth | One should mitigate security risk by employing several defensive layers of secure coding practice, so that if one is exploited – the attacker is still blocked from entry. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance teams should exist to review the code and identify existing or potential flaws. Activities such as fuzz testing, penetration testing and code audits should all be employed. |
| 1. Adopt a Secure Coding Standard | An organization should adopt or develop and apply a coding standard that remains secure for whichever platform or language is utilized during 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-INT50-CPP] | Avoid casting an out-of-range enumeration by checking the given range before casting the enum value. |

| **Noncompliant Code** |
| --- |
| The following code checks for the enum range before casting, which would result in an unspecified value if a value outside the enum’s range is passed. |
| 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 following code first checks that the input passed can be fulfilled within the enum’s range *before* performing the action, thus avoiding an unspecified value. |
| enum EnumType {    First,    Second,    Third  };    void f(**int** intVar) {    if (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** Validate Input Data – An attacker can inject an out-of-range input and cause a buffer overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | N/A |
| Helix QAC | 2021.2 | C++3013 | N/A |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PRQA QA-C++ | 4.4 | 3013 | N/A |
| PVS-Studio | 7.14 | V1016 | N/A |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-DCL52-CPP] | A reference value type should not be cv-qualified when used const, in order to avoid undefined behavior. |

| **Noncompliant Code** |
| --- |
| The following code declares a reference *p* to be a reference of a cv-qualified *char* which results in undefined behavior. |
| #include <iostream>    void f(**char** c) {    const **char** &p = c;    p = 'p'; // Error: read-only variable is not assignable    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| The following code removes the *const* qualifier, which rectifies the solution. |
| #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):** Use Effective Quality Assurance Techniques – The above could result in data integrity violations, so using proper QA testing on code blocks to ensure the integrity of data is necessary here. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 | N/A |
| Helix QAC | 2021.2 | C++0014 | N/A |
| Klocwork | 2021.1 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE | N/A |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-DCL52-a | Never qualify a reference type with ‘const’ or ‘volatile’ |
| Polyspace Bug Finder | R2021a | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| PRQA QA-C++ | 4.4 | 0014 | N/A |
| Clang | 3.9 | N/A | 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 | N/A |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-STR51-CPP] | When creating a string with std, avoid creating it from a null pointer by first checking for null status. |

| **Noncompliant Code** |
| --- |
| The following code creates a result from *std:getenv()* which returns undefined behavior error as it points to a null value. |
| #include <cstdlib>  #include <string>    void f() {    std::string tmp(std::**getenv**("TMP"));    if (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The following code checks for a null status on *std::getenv()* before the object is called. |
| #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):** Architect and Design for Security Policies – referencing a null pointer without checking is bad practice that is likely to lead to vulnerabilities. Designing with this in mind can mitigate attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | assert\_failure | N/A |
| Helix QAC | 2021.2 | C++4770, C++4771, C++4772, C++4773, C++4774 | N/A |
| Klocwork | 2021.1 | 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 | N/A |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-FIO30-C] | In order to avoid injection attacks, exclude user input from formatted strings to mitigate tainted values. |

| **Noncompliant Code** |
| --- |
| The following code allows for untrusted user input to be passed as a format string argument to *fprintf().* |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/    static const **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + sizeof(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }  **fprintf**(stderr, msg);  **free**(msg);  } |

| **Compliant Code** |
| --- |
| The following code solves the above problem by replacing *fprintf()* with a call to *fputs()*, outputting the *msg* to *stderr* without evaluation. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/    static const **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + sizeof(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }  **fputs**(msg, stderr);  **free**(msg);  } |

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

| **Principles(s):** Default Deny – Without excluding user input from formatting, a user can access vulnerable permissions in order to execute arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | N/A | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 6.1p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Compass/ROSE | N/A | N/A | N/A |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| GCC | 4.3.5 | N/A | Can detect violations of this rule when the -Wformat-security flag is used |
| Helix QAC | 2021.2 | C4916, C4917, C4918  C++4916, C++4917, C++4918 | N/A |
| Klocwork | 2021.1 | SV.FMTSTR.GENERIC  SV.TAINTED.FMTSTR | N/A |
| LDRA tool suite | 9.7.1 | 86 D | Partially Implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-FIO30-a  CERT\_C-FIO30-b  CERT\_C-FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant  Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable  Never use unfiltered data from an untrusted user as the format parameter |
| PC-lint Plus | 1.4 | 592 | Partially supported: reports non-literal format strings |
| Polyspace Bug Finder | R2021a | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |
| PRQA QA-C | 9.7 | 4916, 4917, 4918 | N/A |
| PRQA QA-C++ | 4.4 | 4916, 4917, 4918 | N/A |
| PVS-Studio | 7.14 | V618 | N/A |
| Splint | 3.1.1 | N/A | N/A |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-MEM50-CPP] | Ensure reallocation or recycling of freed memory to avoid dangling pointers, which can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The following code has *s* dereferenced after already being deallocated. This could result in a vulnerability that can be exploited by running arbitrary code with a vulnerable permissions process. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| The following code does not deallocate the memory until it’s no longer required. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Ensuring proper memory management to avoid vulnerabilities is a secure coding standard to approach. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use | N/A |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 | N/A |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 6.1p0 | ALLOC.UAF | Use after free |
| Compass/ROSE | N/A | N/A | N/A |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Helix QAC | 2021.2 | C++4303, C++4304 | N/A |
| Klocwork | 2021.1 | UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST | N/A |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ | N/A | N/A | Runtime detection |
| Polyspace Bug Finder | R2021a | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |
| PRQA QA-C++ | 4.4 | 4303, 4304 | N/A |
| PVS-Studio | 7.14 | V586, V774 | N/A |
| Splint | 5.0 | N/A | N/A |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-DCL03-C] | Use static assertions to test constant expressions to identify vulnerabilities using preprocessor conditional statements to avoid the assert being skipped over in runtime execution. |

| **Noncompliant Code** |
| --- |
| The following code executes an *assert()* during runtime within a particular function, which may not trigger if the function is not executed during runtime. |
| #include <assert.h>    struct timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(void) {  **assert**(sizeof(struct timer) == sizeof(unsigned **char**) + sizeof(unsigned **int**) + sizeof(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| The following code uses a preprocessor conditional statement to ensure the *assert()* is evaluated at compile time, resulting in no runtime penalty. |
| struct timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Proper runtime testing allows for vulnerabilities to be discovered and exceptions properly thrown. |
| --- |

**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-DCL03 | N/A |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 6.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE | N/A | N/A | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-ERR51-CPP] | Each exception should be handled during runtime to avoid a sudden and ‘ungraceful’ termination of the program. |

| **Noncompliant Code** |
| --- |
| The following code does not catch exception thrown from *throwing\_fun()* as no matching handler is found. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| The following code handles all exceptions in *main()* which allows for ‘graceful’ management of resources. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    **int** main() {    try {      f();    } catch (...) {      // Handle error    }  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Proper runtime testing allows for vulnerabilities to be discovered and exceptions properly thrown. Handling exceptions falls under this standard. |
| --- |

**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 | N/A |
| Helix QAC | 2021.2 | C++4035, C++4036, C++4037 | N/A |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++test | 2021.1 | 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 | R2021a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| PRQA QA-C++ | 4.4 | 4035, 4036, 4037 | N/A |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-CTR53-CPP] | Ensure that when iterating over a container elements, that the iteration is over a valid range – otherwise results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The following code uses two iterators to point to the same container, but one continues iterating as there are compared for equal value. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<**int**> &c) {    std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| The following code passes the iterations to *std::for\_each()* in proper order. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):** Architect and Design for Security Policies – Ensuring proper iteration over container elements to avoid buffer overflows is a design decision that should be implemented alongside development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | overflow\_upon\_dereference | N/A |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2021.2 | C++3802 | N/A |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.1 | **CERT\_CPP-CTR53-a CERT\_CPP-CTR53-b** | Do not use an iterator range that isn't really a range Do not compare iterators from different containers |
| [PRQA QA-C++](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046345) | 4.4 | 3802 | N/A |
| PVS-Studio | 7.14 | **V539, V662, V789** | N/A |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Class Pointers | [STD-EXP57-CPP] | Avoid referring to objects of an incomplete class type, otherwise known as a forward declaration. |

| **Noncompliant Code** |
| --- |
| The following code deletes a pointer to an incomplete class type, which results in undefined behavior if *Body* has a nontrivial destructor. |
| class Handle {    class Body \*impl;  // Declaration of a pointer to an incomplete class  public:    ~Handle() { delete impl; } // Deletion of pointer to an incomplete class    // ...  }; |

| **Compliant Code** |
| --- |
| The following code deletes the pointer where is already defined. |
| class Handle {    class Body \*impl;  // Declaration of a pointer to an incomplete class  public:    ~Handle();    // ...  };    // Elsewhere  class Body { /\* ... \*/ };    Handle::~Handle() {    delete impl;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Ensuring that objects from incomplete class types are not called is a coding standard that can mitigate program termination, runtime signals, and resource leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | delete-with-incomplete-type | N/A |
| Coverity | 6.5 | DELETE\_VOID | Fully implemented |
| Clang | 3.9 | -Wdelete-incomplete | N/A |
| CodeSonar | 6.1p0 | LANG.CAST.PC.INC | Conversion: pointer to incomplete |
| Helix QAC | 2021.2 | C++3112 | N/A |
| Klocwork | 2021.1 | CERT.EXPR.DELETE\_PTR.INCOMPLETE\_TYPE | N/A |
| LDRA tool suite | 9.7.1 | 169 S, 554 S | Enhanced Enforcement |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-EXP57-a  CERT\_CPP-EXP57-b | Do not delete objects with incomplete class at the point of deletion  Conversions shall not be performed between a pointer to an incomplete type and any other type |
| Parasoft Insure++ | N/A | N/A | Runtime detection |
| Polyspace Bug Finder | R2021a | CERT C++: EXP57-CPP | Checks for conversion or deletion of incomplete class pointer |
| RuleChecker | 20.10 | delete-with-incomplete-type | N/A |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Management | [STD-FIO42-C] | Ensure that files are closed when they are no longer needed, to avoid attackers exploiting and exhausting system resources. |

| **Noncompliant Code** |
| --- |
| The following code does not close the file opened by *fopen()* before the *func()* function returns. |
| #include <stdio.h>    **int** func(const **char** \*filename) {  **FILE** \*f = **fopen**(filename, "r");    if (NULL == f) {      return -1;    }    /\* ... \*/    return 0;  } |

| **Compliant Code** |
| --- |
| The following code closes the file opened by *f* before returning. |
| #include <stdio.h>    **int** func(const **char** \*filename) {  **FILE** \*f = **fopen**(filename, "r");    if (NULL == f) {      return -1;    }    /\* ... \*/    if (**fclose**(f) == EOF) {      return -1;    }    return 0;  } |

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

| **Principles(s):** Architect and Design for Security Policies – Ensuring that file management is implemented alongside development will prevent attackers from taking advantage of system resources. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | N/A | Supported, but no explicit checker |
| CodeSonar | 6.1p0 | ALLOC.LEAK | Leak |
| Compass/ROSE | N/A | N/A | N/A |
| Coverity | 2017.07 | RESOURCE\_LEAK (partial) | Partially implemented |
| Helix QAC | 2021.2 | C2701, C2702, C2703  C++2701, C++2702, C++2703 | N/A |
| Klocwork | 2021.1 | RH.LEAK | N/A |
| LDRA tool suite | 9.7.1 | 49 D | Partially implemented |
| Parasoft C/C++test | 2021.1 | CERT\_C-FIO42-a | Ensure resources are freed |
| PC-lint Plus | 1.4 | 429 | Partially supported |
| Polyspace Bug Finder | R2021a | CERT C: Rule FIO42-C | Checks for resource leak (rule partially covered) |
| PRQA QA-C | 9.7 | 2701, 2702, 2703 | N/A |
| PRQA QA-C++ | 4.4 | 2701, 2702, 2703 | N/A |
| SonarQube C/C++ Plugin | 3.11 | S2095 | N/A |

### 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.]

The automation tools best used for enforcing the coding standards and the new DevSecOps pipeline will be considered during the pre-production phase. Planning will include the consideration of such automated tools, and they will be utilized while building and verifying the integrity of the system. Following, they will remain as steadfast static and runtime testing tools during production timelines of monitoring, detecting and responding to any flaws in the system.

### 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-INT50-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-DCL52-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-STR51-CPP | High | Likely | Medium | P18 | L1 |
| STD-FIO30-C | High | Likely | Medium | P18 | L1 |
| STD-MEM50-CPP | High | Likely | Medium | P18 | L1 |
| STD-DCL03-C | Low | Unlikely | High | P1 | L3 |
| STD-ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| STD-CTR53-CPP | High | Probable | High | P6 | L2 |
| STD-EXP57-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-FIO42-C | Medium | Unlikely | Medium | P4 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encrypting data that is stored on a disk or other drive. If an attacker obtains access to an encrypted drive and doesn’t have the keys, they can’t access it. |
| Encryption at flight | This is the process of encrypting data while being transmitted, this would apply to all data between servers and devices that could be compromised. |
| Encryption in use | This is securing data by ensuring it’s never left ‘unsecure.’ This would include using permission-based roles in order to mitigate attacks on data in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authenticating users via usernames, passwords and other mechanisms applies when utilizing standards of default deny and permission-based coding. |
| Authorization | This follows authentication and is a second layer of authorizing a user for a particular set of tasks and access to relevant data. This also applies to useful coding standard practice. |
| Accounting | Accounting is a method of monitoring resources used during access to the network. This includes data, time, and connections during a particular session. It’s important for analysis and keeping a secure log of all activities by user. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 10/09/2021 | Finished Report | Joseph A. Urso | [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 |