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



# Green Pace Secure Development Policy

Victoria Kaloudis

Southern New Hampshire University

CS-405-R1910

Pr. **Ahlam Alhweiti**

October 11, 2023

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate data from all data sources, especially untrusted ones. Using correct input validation can assist in eliminating most software vulnerabilities. External data sources are typically seen as untrustful. Some examples are command line arguments, network interfaces, environmental variables, and user-controlled files (Seacord et al. 2006). |
| 1. Heed Compiler Warnings | Use the highest warning level available for your compiler when creating code. Utilizing static testing and dynamic testing will ensure the code has no vulnerabilities or security flaws (Seacord et al. 2006). |
| 1. Architect and Design for Security Policies | Create and design software based on your security policies. Be mindful of the software architecture and design you have created in relation to your security policies (Seacord et al. 2006). |
| 1. Keep It Simple | Keep the code and design as simple as possible. This will help you in decreasing the number of errors, vulnerabilities, or security flaws found when configuring the design. More complex designs are more difficult to achieve high levels of quality and assurance (Seacord et al. 2006). |
| 1. Default Deny | Create a protection scheme based on access permissions. Default the access based on whether the user has the correct permission to access (Seacord et al. 2006). |
| 1. Adhere to the Principle of Least Privilege | Be mindful of the process execution using elevated permission. Try to utilize the least set of privileges necessary to complete any job. This will help in reducing the opportunities that attackers have to access elevated privileges (Seacord et al. 2006). |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex subsystems should be sanitized. Attackers might access them through SQL, injection, and command attacks. Some examples of complex subsystems are command shells, relational databases, and commercial off-the-shelf components (Seacord et al. 2006). |
| 1. Practice Defense in Depth | Defense in depth is important to manage risk in your code by creating multiple layers of defense strategies in case one-layer falls, there will be other layers to prevent a security flaw. This will create a treacherous path for attackers when they attack your system (Seacord et al. 2006). |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques support you in identifying and eliminating vulnerabilities in the code. Things such as dynamic analysis, static analysis, fuzz testing, penetration testing, and source code audits can help with determining vulnerabilities in your code (Seacord et al. 2006). |
| 1. Adopt a Secure Coding Standard | Either create a secure coding standard or use one already made for developing your code in the chosen language and platform (Seacord et al. 2006). |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not define a C-style variadic function (Saks et al. 2007). |

| **Noncompliant Code** |
| --- |
| The noncompliant code results in undefined behavior. The code below uses a C-style variadic function for addition of integers, until the value 0 is found. As there is no passing the value of 0, the result will be undefined behavior (Saks et al. 2007). |
| #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** |
| --- |
| A parameter pack is needed to make the code compliant. The pack implements the add() function which prevents the error seen above (Saks et al. 2007). |
| #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):** The data type standard maps to architect and design for security policies and validating input data. The data type standard refers to different types of data such as integers, characters, or strings. These data types are used when designing and creating software. Therefore, the data type standard maps to architect and design for security policies. Also, it is important to ensure proper inputs when validating input data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Function-ellipses | Fully checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL50 |  |
| Clang | 3.9 | Cert-dcl50-cpp | Checked by clang-tidy |
| CodeSonar | 5.4p0 | Lang.STRUCT.ELLIPSIS | Ellipses |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not cast to an out-of-range enumeration value (Weis et al. 2010). |

| **Noncompliant Code** |
| --- |
| This code attempts to check if the value is within acceptable enumeration values (Weis et al. 2010). |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| By adding in code to check if the value is within acceptable range before performing a conversion (Weis et al 2010). |
| 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):** The data value standard maps to both validating input data and architect and design for security policies. It is important to validate the data you input into the system and build secure code. It is important to ensure you are not overflowing the buffer. Building secure architecture with proper security policies are important. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivivion Bauhaus Suite | 6.9.0 | CertC++-INT50 |  |
| Helix QAC | 2021.1 |  |  |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-INT50-a |  |
| PRQA QA-C++ | 4.4 | 3013 |  |
| PVS – Studio | 7.07 | V1016 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator (Pincar et al. 2008). |

| **Noncompliant Code** |
| --- |
| The code leads to buffer overflow as the input is unbounded (Pincar et al. 2008). |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Switching to std::string will help ensure the data is not truncated (Pincar et al. 2008). |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** String correctness maps to validating input data, architect design for security policies, heed compiler warnings, keep it simple, and use effective quality assurance techniques. If the string is not stored with sufficient space for character data, it will result in buffer overflow. Ensuring the system does not overflow is imperative for security. Using secure coding practices can help when designing the architecture for the system. Also, you want to heed compiler warnings as you are building your code. This will help you keep your secure best practices and ensure your software has no issues. Static and dynamic testing are great ways to ensure your code has the proper security and no vulnerabilities in the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 |  | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR31** | Detects calls to unsafe string function that may cause buffer overflow Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **LANG.MEM.BO LANG.MEM.TO MISC.MEM.NTERM BADFUNC.BO.\*** | Buffer overrun Type overrun No space for null terminator A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| [Compass/ROSE](https://www.securecoding.cert.org/confluence/display/seccode/Rose) |  |  | Can detect violations of the rule. However, it is unable to handle cases involving strcpy\_s() or manual string copies such as the one in the first example |
| [Coverity](https://www.securecoding.cert.org/confluence/display/seccode/Coverity) | 2017.07 | **STRING\_OVERFLOW**  **BUFFER\_SIZE**  **OVERRUN**  **STRING\_SIZE** | Fully implemented |
| [Fortify SCA](https://www.securecoding.cert.org/confluence/display/seccode/Fortify) | 5.0 |  |  |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.3 | **C2840,  C5009, C5038**  **C++0145, C++5009, C++5038**  **DF2840, DF2841, DF2842, DF2843, DF2845, DF2846, DF2847, DF2848, DF2930, DF2931, DF2932, DF2933, DF2935, DF2936, DF2937, DF2938** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.3 | **SV.FMT\_STR.BAD\_SCAN\_FORMAT** **SV.UNBOUND\_STRING\_INPUT.FUNC** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152309) | 9.7.1 | **489 S, 109 D, 66 X, 70 X, 71 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR31-a** **CERT\_C-STR31-b** **CERT\_C-STR31-c** **CERT\_C-STR31-d** **CERT\_C-STR31-e** | Avoid accessing arrays out of bounds Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Avoid using unsafe string functions which may cause buffer overflows |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **421, 498** | Partially supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule STR31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr31c.html) | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.26 | [**V518**](https://pvs-studio.com/en/docs/warnings/v518/), [**V645**](https://pvs-studio.com/en/docs/warnings/v645/), [**V727**](https://pvs-studio.com/en/docs/warnings/v727/), [**V755**](https://pvs-studio.com/en/docs/warnings/v755/) |  |
| [Splint](https://www.securecoding.cert.org/confluence/display/seccode/Splint) | 3.1.1 |  |  |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **mem\_access** | Exhaustively verified (see [one compliant and one non-compliant example](https://taas.trust-in-soft.com/tsnippet/t/144ae03a)). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Do not store an already-owned pointer value in an unrelated smart pointer (Ballman et al. 2015). |

| **Noncompliant Code** |
| --- |
| A double-free vulnerability is created when two unrelated pointers are constructed from the same pointer value. This is a memory leak and could allow attackers to write in value in memory spaces (Ballman et al. 2015). |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| This code is using the std::shared\_ptr objects by relating them through copy construction. This helps solve the problem of the memory leak (Ballman et al. 2015). |
| #include <memory>    void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):** SQL Injection maps to the principles architect and design for security policies, sanitize data sent to other systems, and keep it simple. Designing appropriate architecting systems with secure code in mind will help eliminate potential SQL injection. SQL injection can lead to results in undefined behavior making the code more vulnerable to hackers. It is also important to sanitize the data as it goes from one system to the next. With this high severity threat level, sanitization will really help with making sure the code is not exploitable. Also, remember to keep it simple as that can really help others using your code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Dangling\_point\_use |  |
| Helix QAC | 2021.1 |  |  |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-MEM56-a |  |
| PVS-Stutdio | 7.01 | V1006 | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory (Pincar et al. 2008). |

| **Noncompliant Code** |
| --- |
| This code shows S as it is dereferenced and deallocated. In some cases it will result in a write-after-free which can allow attackers to run arbitrary code with the permissions of the process (Pincar et al. 2008). |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| To combat above, the code is dynamically allocating memory until it is no longer required (Pincar et al. 2008). |
| #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):** Memory protection maps to validating input data, architect and design for security policies, use effective quality assurance techniques, and adopt a secure coding standard. It is important to not evaluate a pointer to freed memory as that leads to undefined behavior. Undefined behaviors can lead to easily exploitable data. You should ensure proper inputs are used in data with validating input data, build an architecture to prevent vulnerabilities, make various tests to ensure quality assurance is up to par (things like Unit tests), and to make the system based on a secure coding standard to help prevent vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **dangling\_pointer\_use** | Supported  Astrée reports all accesses to freed allocated memory. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM30** | Detects memory accesses after its deallocation and double memory deallocations |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **ALLOC.UAF** | Use after free |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **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](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.3 | **DF4866, DF4867, DF4868, DF4871, DF4872, DF4873**  **C++3339, C++4303, C++4304** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.3 | **UFM.DEREF.MIGHT** **UFM.DEREF.MUST** **UFM.FFM.MIGHT** **UFM.FFM.MUST** **UFM.RETURN.MIGHT** **UFM.RETURN.MUST** **UFM.USE.MIGHT** **UFM.USE.MUST** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **51 D, 484 S, 112 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-MEM30-a** | Do not use resources that have been freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) |  |  | Runtime analysis |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **449, 2434** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule MEM30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemem30c.html) | Checks for:   * Accessing previously freed pointer * Freeing previously freed pointer   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.26 | [**V586**](https://pvs-studio.com/en/docs/warnings/v586/), [**V774**](https://pvs-studio.com/en/docs/warnings/v774/) |  |
| [Splint](https://wiki.sei.cmu.edu/confluence/display/c/Splint) | 3.1.1 |  |  |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **dangling\_pointer** | Exhaustively verified (see [one compliant and one non-compliant example](https://taas.trust-in-soft.com/tsnippet/t/0d556bb8)). |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **dangling\_pointer\_use** | Supported  Astrée reports all accesses to freed allocated memory. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM30** | Detects memory accesses after its deallocation and double memory deallocations |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression (Seacord et al. 2008). |

| **Noncompliant Code** |
| --- |
| The memory-mapped structure is essential for the code to run correctly. The following code uses the assert() macro to assert whether a property in the memory-mapped structure is true. This leads to a run-time error (Seacord et al. 2008). |
| #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** |
| --- |
| A conditional statement can be used to help evaluate assertions at compile time leading to no runtime penalty (Seacord et al. 2008). |
| 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):** Static assertion maps to use effective quality assurance techniques and practice defense in depth. Static testing is a powerful tool to use while building your code. Static testing checks your code for vulnerabilities and bugs that could easily be exploitable. Using effective quality assurance techniques such as unit testing, can help address all vulnerabilities and create secure code for the system you are building. Also, it is important to practice defense in depth. If one vulnerability is exploited and there is no back up plan, you are in a bad position. However, if you have multiple layers of defense, one vulnerability could be covered by multiple things. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivivion Bauhaus Suite | 6.9.0 | CERT-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 6.0p0 | Customization | Users can implement a custom check that reports uses of the assert() macro |
| Compass/Rose |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion, then the code should use static-assert instead; this assumes ROSE can recognize macro invocation. |
| ÉCLAIR | 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-007-CPP] | Handle all exceptions thrown before main() begins executing (Seacord et al. 2009). |

| **Noncompliant Code** |
| --- |
| A potential exception is not caught in the constructor for S (Seacord et al. 2009). |
| struct S {  S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| Making global S into a local variable with static storage duration allows the exception to be caught (Seacord et al. 2009). |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** Exceptions map to architect and design for security policies, use effective quality assurance techniques, and adopt a secure coding standard. If an exception is thrown, there could be results in abnormal program termination and can lead to security vulnerabilities on the device. Building secure code with appropriate input data is important when designing your architecture. Also, testing code as you go will help with using effective quality assurance techniques. Finally, adopting a secure coding standard will ensure you are following the same practices amongst all projects currently being built and future projects not yet being built. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Potentially throwing static initialization | Partially checked |
| Axivion Bauhaus Suite | 6.9.0 | CERTC++-ERR58 |  |
| Clang | 3.9 | Cert-eer58-cpp | Checked by clang-tidy |
| Helix QAC | 2021.1 |  |  |
| Parasoft C/C++ test | 2020.2 CERT\_CPP\_Err58-a |  | Exceptions shall be raised only after start-up and before termination of the program |
| PRQA QA-C++ | 4.4 | 4634, 4636, 4637, 4639 |  |
| Rule Checker | 20.10 | Potentially throwing static initialization | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Do not invoke virtual functions from constructors or destructors (Dewhurst et al. 2007). |

| **Noncompliant Code** |
| --- |
| The result of this code would throw undefined behavior. Essentially, the code calls to virtual functions from the constructor and destructor (Dewhurt et al. 2007). |
| 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** |
| --- |
| To combat the noncompliant code, the constructors and destructors call a nonvirtual function (Dewhurst et al. 2007). |
| 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):** Object Oriented Programming (OOP) maps to validate input data, architect and design for security policies, and heed compiler warnings. In this example of an OOP coding standard, it is important to ensure you are validating your input data and using proper inputs. Marking a global variable versus a local variable can impact your code by prematurely crashing your system and allowing for attacks on your device. Building the architecture for your system with security policies in mind will help to eliminate this issue. Also, you must heed compiler warnings so as not to run your program with issues in the code. Utilizing static and dynamic testing can ensure you are writing secure code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Virtual call in constructor invalid function pointer | Fully checked |
| Axivion Bauhaus Suite | 3.9 | CertC++-OOP50 |  |
| Clang | 3.9 | Clang analyzer alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| Helix QAC | 2021.1 |  |  |
| LDRA tool suite | 9.7.1 | 467S, 92D | Fully implemented |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-OOP50-a  CERT\_CPP-OOP50-b  CERT\_CPP-OOP50-c  CERT\_CPP-OOP50-d | Avoid calling virtual functions from constructors and destructors. Do not invoke class’s virtual functions from any of its constructors. Do not invoke class’s virtual functions from its destructor. |
| PRQA QA-C++ | 4.4 | 4260, 4261, 4272, 4274, 4275, 4276, 4277, 4278, 4279, 4280, 4281, 4282 |  |
| PVS-Studio | 20.10 | Virtual call-in customer | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | S1699 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Guarantee that library functions do not overflow (Pincar et al. 2009). |

| **Noncompliant Code** |
| --- |
| This code can lead to a buffer overflow by using the std::copy() algorithm. There are no inherent bounds checking. In this code, a vector of integers is copied but does not expand the dest vector leading to buffer overflow (Pincar et al. 2009). |
| #include <algorithm>  #include <vector>    void f(const std::vector<int> &src) {  std::vector<int> dest;  std::copy(src.begin(), src.end(), dest.begin());  // ...  } |

| **Compliant Code** |
| --- |
| This code shows the proper way to use std::copy(). By ensuring the destination container can hold all the elements being copied in it, the solution enlarges the capacity of the vector (Pincar et al. 2009). |
| #include <algorithm>  #include <vector>  void f(const std::vector<int> &src) {  // Initialize dest with src.size() default-inserted elements  std::vector<int> dest(src.size());  std::copy(src.begin(), src.end(), dest.begin());  // ...  } |

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

| **Principles(s):** Containers maps to validating input data, architect and design for security polices, and keep it simple. With this specific standard, if the container is not large enough to hold onto the data, it will cause a buffer overflow. Validating with proper inputs is necessary to ensure you do not overflow the buffer. Also, creating secure code for the architecture of your system is imperative. Building safeguards into the system as you move through your code can save time and money. And finally, make sure to keep it simple and clean. If the code is simple and clean to use, there will most likely be less problematic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) |
| 22.10 | 22.10 | 22.10 | 22.10 |
| **invalid\_pointer\_dereference**  [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | **invalid\_p**ointer\_d**ereference**  [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | **invalid\_pointer\_**dereference [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | **invalid\_pointer\_dereference**  [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) |
| 7.4p0 | 7.4p0 | 7.4p0 | 7.4p0 |
| BADFUNC.BO.\*LANG.MEM.BOLANG.MEM.TBA | BADFUNC.BO.\*LANG.MEM.BOLANG.MEM.TBA | BADFUNC.BO.\*LANG.MEM.BOLANG.MEM.TBA | **BADFUNC.BO.\* LANG.MEM.BO LANG.MEM.TBA** |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initializations | [STD-010-CPP] | Do not modify the standard namespaces |

| **Noncompliant Code** |
| --- |
| Undefined behavior is thrown when the declaration of x is added to the namespace std. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| By placing int x in the namespace nonstd, there are no collisions with other global identifiers. |
| namespace nonstd {  int x;  } |

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

| **Principles(s):** Declarations and Initializations map to architect and design for security policies, keep it simple, use effective quality assurance techniques, and adopt a secure coding standard. Building secure code helps prevent vulnerabilities in the system. Modifying standard namespaces can cause undefined behavior when not utilized correctly which makes the system vulnerable to attacks. Keeping the code simple is a best practice in coding with any language. To avoid this error, compiling the code and performing effective quality assurance tests are key. Using static and dynamic testing can help identify the vulnerabilities and give you tips on fixing it before you run into the issue further down the line. Finally, adopting a secure coding standard is important with preventing security vulnerabilities in your projects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-DCL58 |  |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces ‘std’ and ‘posix’ |
| Polyspace Bug Finder | R2020a | CERT C++: DCL58-CPP | Check for modifications of standard namespaces (rule fully covered) |
| PRQA QA – C++ | 4.4 | 4032, 4035, 4631 |  |

### 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.

Both pre-production and production are extremely important for the DevSecOps process. The pre-production phase should focus on creating a plan to assess priorities and possible regulatory changes, designing an architecture that has secure best practices built in, and building secure infrastructure following those guidelines. The production phase should focus on maintaining the system that was built by performing health checks and penetration testing, monitor and detecting potential vulnerabilities, building a secure response catalogue to vulnerabilities, and to maintain and stabilize those fixes so the system can run without problems.

Automation plays a key role in the DevSecOps lifecycle as it can fast-track many steps of the cycle. For example, static and dynamic testing are great automation tools to utilize during the build and verify and test stages. Static testing tests the code for vulnerabilities without executing the code. The test results will give you a list of vulnerabilities and connect you to the National Institute of Standards and Technology (NIST) database for more details. Dynamic testing tests the code by executing it. It will give you a list of potential vulnerabilities within the system and some tips on how to address them. They should be in use at every step of the process. An example of dynamic testing includes unit testing. Unit testing is a great automated way to test the security level of your program and determine how much of your program is covered.

All coding standards are important for the entire cycle. Validating input data helps with eliminating as many vulnerabilities as possible. Heeding compiler warnings will help with testing and executing the code after every line is programmed. Building a system architecture and designing for security polices will ensure your system is safeguarding private information with no vulnerabilities. Keeping the code simple ensures it’s easy to read and understand so other developers can make changes if need be. Defaulting denial and adhering to the principle of the least privilege are both key in maintaining secure connections after the build phase by restricting access to important features based on who needs the access and who does not. Sanitizing data sent to other systems ensures that data is safely sent to the next step with no vulnerabilities to combat exploits of that data. To help do this, you should practice defense in depth by building multiple layers of defense. It may seem redundant, but if one layer goes down, there will be more layers of defense to take its place. Using effective quality assurance techniques is extremely important. Static and dynamic testing was mentioned quite often in this report as they are great resources for ensuring the program is secure enough to handle exploits of sensitive information. Finally, none of this could be possible without the adoption of a secure coding standard. Having a guideline for the DevSecOps lifecycle will help with designing your code using secure best coding practices. It will help with preventing problems down the line.

### 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 | P12 | L1 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | High | Likely | High | P9 | L2 |
| STD-010-CPP | High | Unlikely | Medium | P6 | 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 in rest refers to encryption of data that is currently being stored (Dinic et al. 2022). This should be used as many people are not using software 24/7. For example, username and password combinations are not used all the time. They are stored in a database in the system. The encrypted data is in rest and called upon when the user wants to log into their account. |
| Encryption at flight | Encryption at flight refers to encryption of data that is currently in movement from one source to the next (Dinic et al. 2022). This should be used to create a secure network to move information from one computer to another computer, or even from network to network. For example, sending an instant message from one coworker to another would use encryption in flight. |
| Encryption in use | Encryption in use refers to the encryption of data that is currently in use (Dinic et al. 2022). This should be used anytime the software is accessed, sent, processed, etc. It is the most vulnerable time for data as it is immediately available. Creating secure code during this stage is critical for protection against threats. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is used to determine who the user is (Fortinet et al. n.d.). When a user registers, or becomes a new user, the username and password are encrypted and saved in a system. The authentication step tests the username and password you type in against the encrypted data to ensure you are who you say you are. Multifactor authentication is a common extra security feature that many companies are using to prevent authentication breaches. |
| Authorization | Authorization refers to the level of authorization that a user has (Fortinet et al. n.d.). Certain users might have different levels of authorization like those in the IT department would have more authorization than a sales rep might have. Having various levels of authorization help make a system more secure as exploiters need to hack more people to determine who has the appropriate authorization. |
| Accounting | Accounting refers to the logs that are kept in the system to determine where and what each user accessed during their connection (Fortinet et al. n.d.). This is especially good for IP address checking. If the system notices an unusual IP address being used, they can let IT know there might be a breach. Also, if a user changes anything in a database, there can be a notification system set up to let IT know. If a hacker tries to decrypt data, IT can get a notification. Even if a file is accessed by an unknown user, having an accounting system will allow for a more secure infrastructure. |

### 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
  + 3 – Architect and design for security policies
    - Building an accounting system into your infrastructure is key for secure coding.
  + 4 – Keep it simple
    - The simpler the code is to implement and understand, the better. Meaning, you want to ensure the operating system logs are not needlessly complicated. Practicing secure coding best practices will help ensure the operating system logs are secure.
  + 5 – Default Deny
    - Deny all users access unless access is given to those who need it. This will keep access to the operating system logs to a smaller group of users and limit the potential for a threat.
  + 6 – Adhere to the principle of least privilege
    - This goes with default deny. The least amount of people that have access to information, the easier it is to pin point vulnerabilities in the system.
  + 8 – Practice defense in depth
    - Creating multiple layers of security for the operating system logs will ensure that it is harder to penetrate the system. If one layer goes down, there should be another layer there to keep the logs secure.
  + 10 – Adopt a secure coding standard
    - Adopting a secure coding standard will ensure that everyone is on the same page when building this system in terms of security.
* Firewall logs
  + 4 – Keep it simple
    - The simpler the code is to implement and understand, the better. Meaning, you want to ensure the firewall logs are not needlessly complicated. Practicing secure coding best practices will help ensure the operating system logs are secure.
  + 5 – Default Deny
    - Deny all users access unless access is given to those who need it. This will keep access to the firewall logs to a smaller group of users and limit the potential for a threat.
  + 6 – Adhere to the principle of least privilege
    - This goes with default deny. The least amount of people that have access to information, the easier it is to pinpoint vulnerabilities in the system.
  + 7 – Sanitize data sent to other systems
    - Sanitizing data sent between systems will help keep the firewall logs secure. By preventing unnecessary data from being transported to another system, you can lower the chances of potential exploitation of data.
  + 8 – Practice defense in depth
    - Creating multiple layers of security for the firewall logs will ensure that it is harder to penetrate the system. If one layer goes down, there should be another layer there to keep the logs secure.
  + 10 – Adopt a secure coding standard
    - Adopting a secure coding standard will ensure that everyone is on the same page when building this system in terms of security.
* Anti-malware logs
  + 4 – Keep it simple
    - The simpler the code is to implement and understand, the better. Meaning, you want to ensure the anti-malware logs are not needlessly complicated. Practicing secure coding best practices will help ensure the operating system logs are secure.
  + 5 – Default Deny
    - Deny all users access unless access is given to those who need it. This will keep access to the firewall logs to a smaller group of users and limit the potential for a threat.
  + 6 – Adhere to the principle of least privilege
    - This goes with default deny. The least amount of people that have access to information, the easier it is to pinpoint vulnerabilities in the system.
  + 7 – Sanitize data sent to other systems
    - Sanitizing data sent between systems will help keep the anti-malware logs secure. By preventing unnecessary data from being transported to another system, you can lower the chances of potential exploitation of data.
  + 8 – Practice defense in depth
    - Creating multiple layers of security for the anti-malware logs will ensure that it is harder to penetrate the system. If one layer goes down, there should be another layer there to keep the logs secure.
  + 10 – Adopt a secure coding standard
    - Adopting a secure coding standard will ensure that everyone is on the same page when building this system in terms of security.

## 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 | 07/17/2023 | Module 3 Milestone | Victoria Kaloudis | **Ahlam Alhweiti** |
| 1.2 | 10/11/2023 | Project 1 | Victoria Kaloudis | **Ahlam Alhweiti** |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

Works Cited

Ballman, A. (2015, June 19). *MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer - SEI CERT C++ Coding Standard - Confluence*. Carnegie Mellon University Software Engineering Insititue. Retrieved September 17, 2023, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM56-CPP.+Do+not+store+an+already-owned+pointer+value+in+an+unrelated+smart+pointer

Dewhurst, S. C. (2007, September 27). *OOP50-CPP. Do not invoke virtual functions from constructors or destructors*. Carnegie Mellon University Software Engineering Institute. Retrieved September 17, 2023, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors>

Dinic, M. (2022, May 2). *A Complete Guide to Encryption — Data at Rest, Data in Motion and Data in Use*. Jatheon. Retrieved October 15, 2023, from <https://jatheon.com/blog/data-at-rest-data-in-motion-data-in-use/>

Pincar, J. (2008a, August 13). *CTR52-CPP. Guarantee that library functions do not overflow*. Carnegie Mellon University Software Engineering Insititute. Retrieved September 17, 2023, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR52-CPP.+Guarantee+that+library+functions+do+not+overflow>

Pincar, J. (2008b, August 13). *MEM50-CPP. Do not access freed memory*. Carnegie Mellon University Software Engineering Insitute. Retrieved September 17, 2023, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

Pincar, J. (2008c, August 13). *STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator*. Carnegie Mellon University Software Engineering Insititute. Retrieved September 17, 2023, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator>

Saks, D. (2007, December 21). *DCL50-CPP. Do not define a C-style variadic function*. Carnegie Mellon University Software Engineering Insititue. Retrieved September 17, 2023, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function

Seacord, R. (2008, January 24). *DCL03-C. Use a static assertion to test the value of a constant expression - SEI CERT C Coding Standard - Confluence*. Carnegie Mellon University Software Engineering Insititute. Retrieved September 17, 2023, from https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression

Seacord, R. (2009a, September 21). *Top 10 Secure Coding Practices*. Carnegie Mellon University Software Engineering Institute. Retrieved September 17, 2023, from <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCommentId=88044413>

Seacord, R. (2009b, December 5). *ERR58-CPP. Handle all exceptions thrown before main() begins executing*. Carnegie Mellon University Software Engineering Institute. Retrieved September 17, 2023, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR58-CPP.+Handle+all+exceptions+thrown+before+main%28%29+begins+executing

Svoboda, D. (2009, January 1). *DCL58-CPP. Do not modify the standard namespaces*. Carnegie Mellon University Software Engineering Institute. Retrieved September 17, 2023, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL58-CPP.+Do+not+modify+the+standard+namespaces

Weis, A. (2010, February 19). *INT50-CPP. Do not cast to an out-of-range enumeration value*. Carnegie Mellon University Software Engineering Institute. Retrieved September 17, 2023, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value>

*What is Authentication, Authorization, and Accounting (AAA)?* (n.d.). Fortinet. Retrieved October 15, 2023, from <https://www.fortinet.com/resources/cyberglossary/aaa-security>