**Dre’ Scheetz**

**CS 405**

**04/18/2024**

**Joe Conlan**



# 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 a critical security practice that verifies the data inputted into a software system meets specific criteria before processing. By validating input data, developers can prevent errors or vulnerabilities caused by invalid or malicious data. This technique ensures that inputs are of the correct type, within the acceptable range, and conform to the system's requirements. Additionally, data sanitization, canonicalization, and normalization are complementary techniques that further enhance the system's security. |
| 1. Heed Compiler Warnings | Paying close attention to compiler warnings is crucial in preventing security vulnerabilities and improving code quality. Setting the compiler to its most sensitive warning level can help catch potential issues in the code. Static and dynamic analysis tools can also be incorporated to add an extra layer of scrutiny, uncovering and rectifying security flaws that may not be immediately evident through compiler warnings alone. |
| 1. Architect and Design for Security Policies | Designing software with security in mind is critical to creating an architecture that inherently supports and enforces security policies. For systems that demand varying privileges based on different operational requirements, it's beneficial to segment the system into separate subsystems. Each subsystem should be designed to operate with only the privileges necessary for its specific function. This approach minimizes the risk of security breaches by limiting the access rights of each component, ensuring that they have only the permissions they need to perform their designated tasks. |
| 1. Keep It Simple | Embracing simplicity in software design is fundamental to enhancing security and reliability. A simpler, more streamlined design reduces the chances of errors during implementation, configuration, and usage. Complexity in design makes it more challenging to identify and fix errors and significantly escalates the effort needed to ensure a robust level of security. Keeping the system's architecture and design straightforward makes it easier to understand, manage, and secure, minimizing potential vulnerabilities and making security mechanisms more effective and less cumbersome to implement. |
| 1. Default Deny | The "default deny" principle is a security posture that advocates blocking access to resources by default and explicitly granting permissions only under defined conditions. This approach ensures that access is based on explicit permission rather than exclusion, significantly enhancing a system's security. By adopting this principle, unauthorized access attempts are automatically denied unless they meet the specific criteria outlined in the security policy. This method minimizes the risk of unintended access, making it a fundamental strategy in designing secure systems. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege mandates that every process in a system operates with the minimum privileges necessary for its function. This minimizes the risk of unauthorized actions by limiting the access rights of each process to only what is essential for its tasks. Additionally, any need for elevated permissions should be temporary and only for the duration required to perform the specific privileged activity. This strategy significantly curtails the chances for attackers to exploit elevated privileges, thereby enhancing the system's overall security by reducing the attack surface available to potential threats. |
| 1. Sanitize Data Sent to Other Systems | It's essential to clean data before transferring it to other systems, mainly when interacting with complex subsystems, such as command interfaces, databases, and off-the-shelf software. These areas can be susceptible to various injection attacks, where attackers manipulate the system to execute unintended actions by exploiting these vulnerabilities. This issue often stems not from the subsystem's inability to validate input but from its lack of understanding of the data's intended use. Thus, the onus is on the process of initiating the call to ensure the data is purified of any elements that could be exploited, effectively minimizing the potential for malicious activities and enhancing the system's security posture. |
| 1. Practice Defense in Depth | Employ a multi-layered security approach to mitigate risks effectively. By adopting various defensive measures, the impact of a breach in one security layer can be contained or neutralized by another, preventing a potential security lapse from escalating into a full-blown vulnerability. For instance, integrating secure coding practices with a secure execution environment can significantly diminish the chance of residual vulnerabilities in the code being exploited once the software is in use. This strategy ensures that even if an attacker breaches one defense line, additional safeguards are in place to protect the system, enhancing overall security resilience. |
| 1. Use Effective Quality Assurance Techniques | Implementing robust quality assurance (QA) methods is crucial for detecting and addressing vulnerabilities within a system. Incorporating practices such as fuzz testing, penetration testing, and thorough source code reviews as integral components of a QA program can significantly enhance identifying and eliminating security weaknesses. Furthermore, seeking independent security evaluations can contribute to developing more secure systems. External reviewers offer a fresh perspective, often crucial for spotting and rectifying erroneous assumptions that internal teams might overlook. This approach not only strengthens the system's security but also ensures a comprehensive evaluation of its resilience against potential threats. |
| 1. Adopt a Secure Coding Standard | Embrace a secure coding framework by creating or applying established secure coding practices tailored to your specific development language and platform. This approach ensures that your coding practices systematically address security considerations, minimizing vulnerabilities from the outset and enhancing the overall security posture of your software projects. Adopting such standards is a proactive measure toward building more secure and robust applications. |

### 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 | Due to the versatility of C++ functions, especially those that deal with variadic functions, there is a higher risk of undefined behavior due to type mismatches or incorrect argument counts. Variadic functions can accept various numbers of arguments, but they lack built-in mechanisms for verifying the type and count of these arguments. This absence can lead to errors that are difficult to detect at compile time, resulting in runtime vulnerabilities. To mitigate such risks, it is advisable to use safer alternatives like function parameter packs, which ensure type safety and reduce the chances of unexpected behavior. This approach enhances the overall robustness and reliability of the code. The standard uses DCL50-CPP as a reference. |

| **Noncompliant Code** |
| --- |
| This code snippet illustrates a function designed to sum integers using C-style variadic arguments. It relies on a termination condition that may never be met if the caller forgets to include a trailing 0, leading to undefined behavior. Furthermore, it lacks type safety for the arguments being passed, potentially causing further undefined behavior if non-integer types are provided. |
| #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** |
| --- |
| The example demonstrates a type-safe method for summing a series of integers. Leveraging **std::enable\_if** guarantees that the function is only instantiated with integral types, thus avoiding undefined behavior and making the program well-formed even with a variable number of parameters.  In the second example, the method expands the function parameter pack within a braced initializer list, ensuring type safety without necessitating recursive function calls. By leveraging the constraints of braced initializer lists, which disallow narrowing conversions, this approach maintains strict type safety and simplifies the summation logic. |
| #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...);  }  #include <type\_traits>    **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg i, Arg j, Ts... all) {  **int** values[] = { j, all... };  **int** r = i;  **for** (auto v : values) {      r += v;    }  **return** r;  } |

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

| **Principles(s):** Validate Input Data, Keep it Simple, Adhere to the Principle of Least Privilege, Adopt a Secure Coding Standard  Validating input data ensures that arguments passed to variadic functions are correct and safe; simplifying the design using safer alternatives like function parameter packs reduces complexity and potential errors. Adhering to the principle of least privilege restricts functions to necessary data, enhancing security while adopting a secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High (P12) | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **function-ellipsis** | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.ELLIPSIS** | Ellipsis |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **41 S** | Fully Implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL50-a** | Functions shall not be defined with a variable number of arguments |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | This standard (INT50-CPP) underscores the importance of ensuring that data values remain within their defined range, especially when cast to enumeration types. Enumerations, with their potentially fixed or unfixed underlying types, present a challenge as they can hold values beyond those explicitly defined by their enumerators. Adhering to this standard minimizes the risk of undefined behavior by enforcing type safety and value correctness in conversions, which is crucial for maintaining the reliability and predictability of the code. This approach guards against the accidental introduction of unspecified values, which could lead to errors or vulnerabilities in software applications. |

| **Noncompliant Code** |
| --- |
| In this example, the integer value is cast to the **EnumType** before verifying whether it falls within the enumeration's range. This premature casting can result in the variable **enumVar** holding an unspecified value if **intVar** is outside the [0..3] range, leading to potentially undefined behavior in subsequent operations. |
| **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 first example demonstrates a pre-casting check to ensure that the integer value falls within the permissible range of the enumeration. This guarantees that the conversion to **EnumType** does not yield an unspecified value, preventing undefined behavior.  Using a scoped enumeration ensures that all values are explicitly cast from the integer type to the enumeration type without risking the introduction of unspecified values. Scoped enumerations provide better type safety and namespace control, reducing the likelihood of errors.  The final solution applies a fixed underlying type to an unscoped enumeration, allowing any integer value to be safely converted to the enumeration type. It enhances type safety by explicitly stating the underlying type, thus preventing unintended values. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  }  **enum** **class** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);  }  **enum** EnumType : **int** {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** Validate Input Data, Keep it Simple, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard  For the standard focusing on ensuring data values stay within their defined ranges when cast to enumeration types, fundamental principles include **Validate Input Data**, ensuring values are within acceptable limits before conversion, and **Adopting a Secure Coding Standard** like STD-002-CPP itself, promoting adherence to best practices for type safety and value correctness. **Keep It Simple** reduces risks by avoiding complex conversions that may introduce errors. Use**Effective Quality Assurance Techniques** like code reviews and static analysis to ensure the implementations prevent out-of-range values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **cast-integer-to-enum** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-INT50-a** | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: INT50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcint50cpp.html) | Checks for casting to out-of-range enumeration value (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | String correctness ensures the safe handling and manipulation of strings, a critical aspect of preventing security vulnerabilities like buffer overflows. STR50-CPP highlights the importance of understanding and correctly implementing their manipulation to avoid errors. Ensuring that buffers are adequately sized to hold their data, including any necessary terminators, is fundamental to secure coding practices. By favoring mechanisms that inherently manage size and termination (like **std::string** in C++), developers can significantly mitigate the risk of buffer overflow vulnerabilities, which have historically been a common source of security flaws in software. |

| **Noncompliant Code** |
| --- |
| The first example demonstrates the risk of buffer overflow by reading input into a statically sized buffer without checking the length of the input, potentially exceeding the buffer's capacity.  In this scenario, while an attempt is made to limit input size using **std::cin.width(12)**, the setting applies only to the next input operation, leading to a risk of buffer overflow for subsequent reads.  This code uses **std::basic\_istream::read()** to fill a buffer but fails to ensure the buffer is null-terminated, risking undefined behavior when constructing a **std::string** from the buffer. |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  }  #include <iostream>    **void** f() {  **char** bufOne[12];  **char** bufTwo[12];    std::cin.width(12);    std::cin >> bufOne;    std::cin >> bufTwo;  }  #include <fstream>  #include <string>    **void** f(std::istream &in) {  **char** buffer[32];  **try** {      in.read(buffer, **sizeof**(buffer));    } **catch** (std::ios\_base::failure &e) {      // Handle error    }      std::string str(buffer);    // ...  } |

| **Compliant Code** |
| --- |
| This solution leverages the **std::string** class for input operations, inherently managing memory and avoiding buffer overflow risks associated with fixed-size buffers.  Constructing the **std::string** object with the exact number of characters read prevents issues related to lack of null termination and mitigates the risk of exceeding buffer size. |
| #include <iostream>  #include <string>    **void** f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  }  #include <fstream>  #include <string>    **void** f(std::istream &in) {  **char** buffer[32];  **try** {      in.read(buffer, **sizeof**(buffer));    } **catch** (std::ios\_base::failure &e) {      // Handle error    }    std::string str(buffer, in.gcount());    // ...  } |

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

| **Principles(s):** Validate Input Data, Heed Compiler Warnings, Keep it Simple, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard.  Validate Input Data ensures thorough checking of string inputs to avoid errors; Heed Compiler Warnings catches potential string manipulation issues early; Keep It Simple promotes the use of std::string to manage memory safely and reduce complexity; Use Effective Quality Assurance Techniques applies rigorous testing, including static and dynamic analysis, to detect vulnerabilities; and Adopt a Secure Coding Standard, like STR50-CPP, guides secure string handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO LANG.MEM.TO** | No space for null terminator  Buffer overrun Type overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-STR50-b** **CERT\_CPP-STR50-c** **CERT\_CPP-STR50-e** **CERT\_CPP-STR50-f** **CERT\_CPP-STR50-g** | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: STR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr50cpp.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. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Ensuring the safety of string data transmitted to various complex systems is paramount to negating SQL injection, command injection, and similar security threats. Such data might include characters or sequences that, if not properly handled, could inadvertently trigger commands, leading to security vulnerabilities. This necessitates a rigorous data sanitization process to neutralize potential threats posed by these inputs. The examples below (STR02-C. Sanitize data passed to complex subsystems) highlight the critical need for stringent input validation and sanitization practices. |

| **Noncompliant Code** |
| --- |
| The first snippet demonstrates a precarious practice of directly incorporating user-supplied data into a system command, exposing the system to command injection attacks if the input includes shell commands or metacharacters.  In the second, data from an unvalidated source is fed directly into a system call, allowing attackers to manipulate the input to execute unauthorized commands. This illustrates a classic case of argument injection. |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer);  (**void**) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")),    0); |

| **Compliant Code** |
| --- |
| Employing an allowlist strategy filters user input to include only pre-approved characters. This method effectively neutralizes the injection threat by ensuring the data's integrity and safety. The second technique mitigates the risk of argument injection by precluding the interpretation of user input as command-line options. Introducing a "--" argument before processing user data seals off avenues potentially exploited for unauthorized command execution, thereby upholding the system's security. |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  }  (**void**) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")), 0); |

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

| **Principles(s):** Validate Input Data, Sanitize Data Sent to Other Systems, Practice Defense in Depth, Use Effective Quality Assurance Techniques  Validate Input Data to ensure inputs do not contain malicious SQL commands, Sanitize Data Sent to Other Systems to clean inputs before they interact with databases, Practice Defense in Depth by implementing multiple security layers to catch SQL injection attempts, and Use Effective Quality Assurance Techniques like testing and audits to identify vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 |  | Supported by stubbing/taint analysis |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | MEM50-CPP offers a great example of how memory protection in C++ is critical to prevent undefined behavior that can lead to security vulnerabilities, such as exploiting dangling pointers after memory deallocation. Accessing memory that has been freed causes undefined behavior because the memory manager may reallocate this memory, making its content unpredictable. Proper management of memory allocation and deallocation, along with the careful handling of pointers, is essential to avoid creating dangling pointers and to ensure the safety and integrity of memory operations. By adhering to best practices and utilizing features of modern C++, such as smart pointers and automatic storage duration, developers can mitigate risks associated with improper memory management. |

| **Noncompliant Code** |
| --- |
| The first code block shows an object being used after it has been deleted, leading to undefined behavior because the pointer becomes a dangling pointer after the delete operation.  The program attempts to use a pointer to a dynamically allocated memory after the memory has been deallocated by the destructor of a **std::unique\_ptr**, which can result in undefined behavior due to accessing deallocated memory. |
| [#include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  }  #include <iostream>  #include <memory>  #include <cstring>    **int** main(**int** argc, **const** **char** \*argv[]) {  **const** **char** \*s = "";  **if** (argc > 1) {  **enum** { BufferSize = 32 };  **try** {        std::unique\_ptr<**char**[]> buff(**new** **char**[BufferSize]);        std::**memset**(buff.get(), 0, BufferSize);        // ...        s = std::**strncpy**(buff.get(), argv[1], BufferSize - 1);      } **catch** (std::bad\_alloc &) {        // Handle error      }    }      std::cout << s << std::endl;  } |

| **Compliant Code** |
| --- |
| This approach ensures that the dynamically allocated object is used within its lifetime and is deleted only after its use is complete, preventing undefined behavior related to dangling pointers.  Utilizing automatic storage duration for objects that do not need to exist beyond the scope of a function is a safer alternative to dynamic allocation, as it automatically manages the object's lifetime and eliminates the risk of dangling pointers. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  }  **struct** S {  **void** f();  };    **void** g() {    S s;    // ...    s.f();  } |

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

| **Principles(s):** Heed Compiler Warnings, Keep it Simple, Practice Defense in Depth, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard  Heed Compiler Warnings to identify potential memory management issues, Keep It Simple to minimize errors in handling memory, Practice Defense in Depth for layered memory safety checks, Use Effective Quality Assurance Techniques such as static and dynamic analysis to detect memory-related vulnerabilities, and Adopt a Secure Coding Standard following guidelines like MEM50-CPP for best practices in memory management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Klocwork) | 2024.1 | **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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **483 S, 484 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MEM50-a** | Do not use resources that have been freed |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: MEM50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem50cpp.html) | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | The assertion standard plays a crucial role in identifying and fixing software defects early in development. This significantly reduces the potential for vulnerabilities. Although runtime assertions (assert()) are helpful for dynamic diagnostics, they are limited by runtime overheads and their behavior to terminate the program upon failure. Therefore, runtime assertions are unsuitable for scenarios where continuous operation is critical, such as in server environments or embedded systems. Static assertions (static\_assert), introduced in C++11, provide a compile-time alternative that eliminates runtime costs and ensures that specific conditions are met before the program runs. Static assertions improve code safety and integrity without impacting performance by leveraging compile-time evaluations. This makes them an invaluable asset in a developer's toolkit for maintaining code correctness. It's worth noting that while example DCL03-C is not in C++, it's still applicable. |

| **Noncompliant Code** |
| --- |
| The following example demonstrates the use of a runtime assertion to check the size of a structure, a check that is ideally performed at compile time. The limitation here is the runtime evaluation, which means the check only occurs if the code path is executed and could terminate the program if the assertion fails. |
| #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** |
| --- |
| Using a preprocessor directive to perform a compile-time check eliminates runtime overhead. It ensures that the structure's size is validated before the program is compiled, offering a proactive approach to identifying potential memory layout issues.  This example employs a static assertion to verify the size of a structure at compile time, effectively preventing the compilation of code with incorrect assumptions regarding memory layout. This method ensures early detection of errors without impacting runtime performance or behavior, thus enhancing code reliability and safety. |
| **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  #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    static\_assert(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**),                "Structure must not have any padding"); |

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

| **Principles(s):** Heed Compiler Warnings, Keep it Simple, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard  Heeding Compiler Warnings is crucial as assertions, especially static ones, allow preemptive error detection during compilation, enhancing early vulnerability identification. Keeping It Simple helps by using assertions to clarify expected behavior in the code, reducing complexity and improving manageability. Using Effective Quality Assurance Techniques makes assertions a built-in testing tool that verifies assumptions continuously during development, aiding in early problem detection. Adopting a Secure Coding Standard is supported by assertions that ensure the code adheres to predefined safety and functionality standards before execution, thus embedding security into the development lifecycle. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Ensuring software systems remain dependable and resilient, especially in unforeseen circumstances, requires a solid framework for error handling. Exceptions provide a structured approach for signaling and managing errors, allowing systems to recover and gracefully adapt to errors. One proactive strategy for enhancing system dependability is incorporating runtime-constraint handlers, as outlined in ERR03-C. These handlers ensure uniform error handling throughout all system components, ultimately contributing to reliability, safety, and security. By explicitly managing runtime constraints, developers can prevent undefined behaviors and guarantee that systems continue to operate safely, even when encountering errors that would otherwise jeopardize system integrity. |

| **Noncompliant Code** |
| --- |
| In this example, using strcpy\_s without a custom runtime-constraint handler can result in inconsistent runtime error handling across different environments, possibly leading to unexpected program termination. |
| errno\_t function(**char** \*dst1, **size\_t** size){  **char** src1[100] = "hello";    **if** (strcpy\_s(dst1, size, src1) != 0) {  **return** -1;    }    /\* ... \*/  **return** 0;  } |

| **Compliant Code** |
| --- |
| By explicitly setting a runtime-constraint handler with **set\_constraint\_handler\_s**, this solution ensures consistent behavior in response to runtime-constraint violations, enhancing the system's reliability and predictability across different implementations.  For environments such as Visual Studio, setting an invalid parameter handler with **\_set\_invalid\_parameter\_handler** provides a mechanism to handle errors consistently. This approach is tailored to the development environment, ensuring errors are managed effectively and maintaining system integrity. |
| constraint\_handler\_t handle\_errors(**void**) {    /\* Handle runtime-constraint error \*/  }    /\* ... \*/    set\_constraint\_handler\_s(handle\_errors);    /\* ... \*/    /\* Returns zero on success \*/  errno\_t function(**char** \*dst1, **size\_t** size){  **char** src1[100] = "hello";    **if** (strcpy\_s(dst1, size, src1) != 0) {  **return** -1;    }    /\* ... \*/  **return** 0;  }  \_invalid\_parameter\_handler handle\_errors(  **const** **wchar\_t**\* expression,  **const** **wchar\_t**\* function,  **const** **wchar\_t**\* file,     unsigned **int** line,  **uintptr\_t** pReserved  ) {    /\* Handle invalid parameter \*/  }    /\* ... \*/    \_set\_invalid\_parameter\_handler(handle\_errors)    /\* ... \*/    errno\_t function(**char** \*dst1, **size\_t** size) {  **char** src1[100] = "hello";    **if** (strcpy\_s(dst1, size, src1) != 0) {  **return** -1;    }    /\* ... \*/  **return** 0;  } |

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

| **Principles(s):** Heed Compiler Warnings, Keep it Simple, Practice Defense in Depth, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard  Heeding Compiler Warnings ensures potential exception-handling errors are addressed during development, preventing runtime issues. Keeping It Simple facilitates straightforward and maintainable error management through exceptions, enhancing code clarity and security. Practicing Defense in Depth involves using exceptions as part of a comprehensive error-handling strategy to manage errors effectively and ensure system stability. Using Effective Quality Assurance Techniques involves rigorous testing of exception handling to confirm all potential error paths are safely managed. Adopting a Secure Coding Standard ensures that exceptions are handled according to best practices, contributing to the reliability and security of the system. |
| --- |

**Threat Level**

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

**Automation [Not Applicable]**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| N.A. |  |  |  |
| N.A. |  |  |  |
| N.A. |  |  |  |
| N.A. |  |  |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Parameter Validation | STD-008-CPP | Parameter validation is a critical aspect of software development that ensures a system's overall robustness, security, and reliability. In the context of programming languages like C and C++, which rely heavily on manual memory management and low-level access, validating input parameters becomes even more crucial. By validating parameters within the called function (callee), developers can encapsulate the logic for checking validity in one place, simplifying code maintenance, and ensuring consistency across the application. This approach also helps prevent many errors and vulnerabilities, from simple logic bugs to severe security flaws like buffer overflows and null pointer dereferences as seen in API00-C. |

| **Noncompliant Code** |
| --- |
| The code snippet you provided demonstrates two functions, setfile() and usefile(), that operate on a global file pointer without validating its integrity. The absence of checks for the file's validity may lead to undefined behavior or security vulnerabilities, depending on how myFile is used within usefile(). |
| /\* Sets some internal state in the library \*/  **extern** **int** setfile(**FILE** \*file);    /\* Performs some action using the file passed earlier \*/  **extern** **int** usefile();    **static** **FILE** \*myFile;    **void** setfile(**FILE** \*file) {      myFile = file;  }    **void** usefile(**void**) {      /\* Perform some action here \*/  } |

| **Compliant Code** |
| --- |
| The example enhances the program's reliability and security, necessary validations must be added to the setfile() function. This involves checking that the file pointer is not null and is in a valid state before proceeding, preventing potential misuse or exploitation. Similarly, usefile() should ensure myFile is not null before use, maintaining the integrity of the library's operations and protecting against invalid state usage. |
| /\* Sets some internal state in the library \*/  **extern** errno\_t setfile(**FILE** \*file);    /\* Performs some action using the file passed earlier \*/  **extern** errno\_t usefile(**void**);    **static** **FILE** \*myFile;    errno\_t setfile(**FILE** \*file) {  **if** (file && !**ferror**(file) && !**feof**(file)) {      myFile = file;  **return** 0;    }      /\* Error safety: leave myFile unchanged \*/  **return** -1;  }    errno\_t usefile(**void**) {  **if** (!myFile) **return** -1;        /\*       \* Perform other checks if needed; return       \* error condition.       \*/        /\* Perform some action here \*/  **return** 0;  } |

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

| **Principles(s):** Validate Input Data, Keep it Simple, Practice Defense in Depth, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard  Validate Input Data ensures that parameters meet predefined criteria, thus blocking erroneous or malicious data that could lead to vulnerabilities like buffer overflows. Keep It Simple supports consolidating validation logic within the callee to enhance code clarity and maintainability, minimizing the potential for security flaws. Practice Defense in Depth positions parameter validation as a critical frontline defense, preventing harmful inputs from penetrating deeper into the system. Use Effective Quality Assurance Techniques involves rigorous testing of validation routines to confirm their effectiveness against exploitation attempts. Lastly, Adopt a Secure Coding Standard emphasizes the importance of following best practices for input validation, reinforcing the application's defenses against common and emerging threats. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | High | Medium (P2) | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.UPD** | Unchecked parameter dereference |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-API00-a** | The validity of parameters must be checked inside each function |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **413, 613, 668** | Partially supported: reports use of null pointers including function parameters which are assumed to have the potential to be null |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.30 | [**V781**](https://pvs-studio.com/en/docs/warnings/v781/) | N.A. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Compile Cleanly | STD-009-CPP | Ensuring clean code compilation, free from warnings and errors, is crucial for maintaining high code quality and reliability standards. Compilers play an essential role in enforcing language standards and flagging potential issues in code, such as syntax errors, type mismatches, and language constraint violations. Developers should set the highest warning level and address all diagnostic messages during code compilation to quickly catch and resolve potential issues. By following this standard, developers can proactively ensure that their software is robust, portable, and maintainable, as exemplified by MSC00-C. |

| **Noncompliant Code** |
| --- |
| An example of improper use of #pragma warning directives for managing compiler warnings in Windows is when the code disables specific warnings and then attempts to reset them to their default behavior, which could inadvertently alter their previous state and obscure potential issues in flagged code sections. |
| #pragma warning(disable:4705)  #pragma warning(disable:4706)  #pragma warning(disable:4707)  /\* Unnecessarily flagged code \*/  #pragma warning(default:4705)  #pragma warning(default:4706)  #pragma warning(default:4707) |

| **Compliant Code** |
| --- |
| To address this, developers should use #pragma warning(push) and #pragma warning(pop) to save and restore the compiler's warning state around code blocks that may trigger unnecessary warnings. This approach ensures that modifications to the warning state remain localized and do not affect compilation settings outside of the intended scope, maintaining the clarity and integrity of the warning system. |
| #pragma warning(push)  #pragma warning(disable:4705)  #pragma warning(disable:4706)  #pragma warning(disable:4707)  /\* Unnecessarily flagged code \*/  #pragma warning(pop) |

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

| **Principles(s):** Heed Compiler Warnings, Keep It Simple, Use Effective Quality Assurance, Adopt a Secure Coding Standard  Heed Compiler Warnings, setting the compiler to its highest warning level to proactively catch and resolve potential issues, enhancing both security and code quality; Keep It Simple, for straightforward and less complex code, reducing the likelihood of hidden bugs and vulnerabilities; Use Effective Quality Assurance Techniques, ensuring through practices like static analysis that the code not only compiles without warnings but also meets high-quality standards; and Adopt a Secure Coding Standard, which aligns coding practices with established secure guidelines to prevent common errors and enhance overall security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium (P8) | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **BUILD.WALL**  **BUILD.WERROR** | Not All Warnings Are Enabled  Warnings Not Treated As Errors |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.30 | [**V665**](https://pvs-studio.com/en/docs/warnings/v665/) | N.A. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151949) | 3.11 | [**S1762**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-1762)  [**S973**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-973) | Warns when the default warning specifier is used with  #pragma warning.  Requires documentation of #pragma uses |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | STD-010-CPP | Creating files with appropriate access permissions is crucial to ensure data integrity and privacy in software systems, as highlighted in FIO06-C. This practice is essential when files contain sensitive or critical information. The default behavior of file creation functions assigns implementation-defined or overly permissive access rights, which poses a significant security risk. By enforcing stricter access controls explicitly at creation, software developers can prevent unauthorized access and potential data breaches. This approach aligns with broader security best practices, emphasizing limiting access rights to only those necessary for operational requirements. Additionally, it addresses the potential for security vulnerabilities arising from dependence on default system behaviors or configurations, such as the process's umask in POSIX-compliant systems. Implementing this standard contributes to developing secure, reliable, and trustworthy software applications. |

| **Noncompliant Code** |
| --- |
| An example of using fopen() for file creation without explicitly defining access permissions can lead to files being created with default or overly permissive access rights, inadvertently exposing them to unauthorized access. |
| **char** \*file\_name;  **FILE** \*fp;    /\* Initialize file\_name \*/    fp = **fopen**(file\_name, "w");  **if** (!fp){    /\* Handle error \*/  } |

| **Compliant Code** |
| --- |
| One compliant solution is to use the function fopen\_s(), which allows for creating files with restricted permissions. By omitting the 'u' mode character, the file is created with permissions that limit access to other users, enhancing the file's security.  Another solution is to use the fopen() function, which requires the developer to specify file access permissions explicitly at the time of creation. This approach addresses the risk of creating files with overly permissive access rights, ensuring that the file's permissions are appropriately restrictive. |
| **char** \*file\_name;  **FILE** \*fp;    /\* Initialize file\_name \*/    errno\_t res = fopen\_s(&fp, file\_name, "wx");  **if** (res != 0) {    /\* Handle error \*/  }  **char** \*file\_name;  **int** file\_access\_permissions;    /\* Initialize file\_name and file\_access\_permissions \*/    **int** fd = open(    file\_name,    O\_CREAT | O\_WRONLY,    file\_access\_permissions  );  **if** (fd == -1){    /\* Handle error \*/  } |

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

| **Principles(s):** Default Deny, Adhere to the Principle of Least Privilege, Sanitize Data Sent to Other Systems, Practice Defense in Depth  Default Deny restricts file access by default, explicitly allowing only necessary permissions, which is crucial for preventing unauthorized access. **Adhere to the Principle of Least Privilege:** Apply minimal necessary access rights, reducing the risk of data breaches. **Sanitize Data Sent to Other Systems,** though typically focused on data transfer, underscores the importance of safeguarding file contents and metadata to prevent vulnerabilities**. Practice Defense in Depth** integrates strict file access controls within a broader security framework, providing multiple layers of protection to secure data against various threats. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | High | High (P4) | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **(customization)** | CodeSonar's custom checking infrastructure allows users to implement checks such as the following.   * A check for all uses of fopen(). * A check for calls to open() with only two arguments. * A check for calls to open() where the third argument does not satisfy some specified requirement. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.1 | **C5013** | N.A. |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Enhanced Enforcement |

### 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.

Automated tools should be integrated at various stages of the DevSecOps cycle to enforce security standards in Green Pace's existing DevOps process. During the Assess and Plan phase, automated security analysis tools should evaluate threats and regulatory changes to prioritize security tasks. Tools that enforce security best practices, such as OWASP's recommendations, should be incorporated in the design phase. The Build stage should include automated code scanning against secure coding standards, ensuring only trusted repositories are used. Verification and Testing should be automated for vulnerability scanning and integrity checks with digitally signed code. As you transition to pre-production, use scripts for automatic deployment, security configuration, and penetration testing. Implement real-time monitoring and detection in Production with automated log collection and intrusion detection systems. Finally, ensure that automated mechanisms are in place to maintain system stability and respond to incidents, including automatic rollback to a known good state if an attack is detected. By embedding these automated checks and tools into the CI/CD pipeline, Green Pace will enhance its security posture and ensure ongoing compliance with established standards.

### 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 | Probable | Medium | High (P12) | 1 |
| STD-002-CPP | Medium | Unlikely | Medium | Medium (P4) | 3 |
| STD-003-CPP | High | Likely | Medium | High (P18) | 1 |
| STD-004-CPP | High | Likely | Medium | High (P18) | 1 |
| STD-005-CPP | High | Likely | Medium | High (P18) | 1 |
| STD-006-CPP | Low | Unlikely | High | Medium (P1) | 3 |
| STD-007-CPP | Low | Unlikely | Medium | Low (P2) | 3 |
| STD-008-CPP | Medium | Unlikely | High | Medium (P2) | 3 |
| STD-009-CPP | Medium | Probable | Medium | Medium (P8) | 2 |
| STD-010-CPP | Medium | Probable | High | High (P4) | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest involves securing data stored on any physical or digital media. This policy mandates that all sensitive or confidential data, such as personal identifiable information (PII), financial records, or operational data, must be encrypted using strong encryption standards like AES-256 when stored. This ensures that data is unreadable to unauthorized individuals even if they gain physical access to the storage media. It is crucial to comply with data protection regulations and protect corporate data from theft or accidental exposure. |
| Encryption in flight | Encryption in flight ensures that data transmitted over a network is protected against interception and tampering. According to this policy, all data must be encrypted using secure protocols such as TLS/SSL or IPSec when transmitted over public or untrusted networks. This policy applies to all data exchanges, including emails, files, and other data communications, to prevent unauthorized access and maintain the confidentiality and integrity of data during transmission. |
| Encryption in use | Encryption protects data that is actively being processed by applications and systems. This policy requires that sensitive data used within software environments be encrypted to prevent unauthorized access, especially in shared or cloud environments. Techniques like homomorphic encryption or secure enclaves (e.g., Intel SGX) safeguard data even when it is being processed, mitigating risks from potential vulnerabilities within the processing environment or unauthorized system access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication confirms the identity of users or entities trying to access system resources. This policy mandates the use of multi-factor authentication (MFA) for all access to organizational IT resources, integrating something the user knows (password), something the user has (security token), and something the user is (biometric verification). It ensures that only authorized users can access sensitive information and systems, significantly reducing the risk of breaches due to stolen or weak credentials. |
| Authorization | Authorization determines and enforces what authenticated users can do within a system. According to this policy, user permissions should be strictly based on their organizational role, employing a role-based access control (RBAC) system. This ensures users can access only the resources necessary for their job functions, minimizing the potential for data leakage and enhancing security by implementing the principle of least privilege. |
| Accounting | Accounting involves tracking and recording user activities and system access, creating an audit trail for security and operational monitoring. This policy requires that all user actions and system interactions be logged in a secure, non-alterable format. It applies to all systems and networks, providing crucial data that helps understand access patterns, detect potential security incidents, and fulfill compliance requirements for auditability. |

**\***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

| Coding Standard | Rule | Principles |
| --- | --- | --- |
| Data Type | STD-001-CPP | 1, 4, 6, 10 |
| Data Value | STD-002-CPP | 1, 4, 9, 10 |
| String Correctness | STD-003-CPP | 1, 2, 4, 9, 10 |
| SQL Injection | STD-004-CPP | 1, 7, 8, 9 |
| Memory Protection | STD-005-CPP | 2, 4, 8, 9, 10 |
| Assertions | STD-006-CPP | 2, 4, 9, 10 |
| Exceptions | STD-007-CPP | 2, 4, 8, 9, 10 |
| Parameter Validation | STD-008-CPP | 1, 4, 8, 9, 10 |
| Compile Cleanly | STD-009-CPP | 2, 4, 9, 10 |
| Input Output | STD-010-CPP | 5, 6, 7, 8 |

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.5 | 03/31/2024 | Principle Section Completed, Standards and Descriptions added. | Dre’ Scheetz | [Insert text.] |
| 2.0 | 04/18/2024 | Completed Template | Dre’ Scheetz | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

Reference

*Top 10 Secure Coding Practices - CERT Secure Coding - Confluence*. (n.d.). https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices